1994

Linear clustering in database systems

Manish Vijay Bhutkar
University of Wollongong

LINEAR CLUSTERING IN DATABASE SYSTEMS

A thesis submitted in partial fulfilment of the
requirements for the award of the degree
Masters of Science (Honours)
(Computing Science)

from

THE UNIVERSITY OF WOLLONGONG

by

Manish Vijay Bhutkar, B.C.S, M.C.S.

Department of Computer Science

July 15th, 1994

ABSTRACT

A number of non-standard database applications need to store and manipulate the representations of two-dimensional real world objects. Efficient support of these applications requires us to abandon the traditional database models and to develop specialised data structures that satisfy the needs of individual applications. Recent investigations in the area of data structures for spatial databases have produced a number of specialised data structures like quad trees, K-D-B trees, R-trees etc. All these techniques try to improve access to data through various indices that reflect the partitions of two-dimensional search space and the geometric properties of represented objects. The other way to improve efficiency is based on linear clustering of disk areas that store information about the objects residing in respective partitions. A number of techniques for linear clustering of objects have been proposed so far. They include Gray curve, Hilbert curve, z-scan curve and snake curve. Unfortunately all these approaches assumed uniform and regular partitions of the search space. A number of interesting questions arise: Are the linear clustering techniques developed so far good for non-uniform partitions of space? Is there a specialised linear clustering technique that provides better results for non-uniform partitions of space?

The main objective of this thesis is to provide an algorithm for linear clustering of non-uniform partitions of two-dimensional space. An evaluation of existing linear clustering techniques for uniform partitions of space is also made.

Acknowledgments

For his guidence in research and in the preparation of this thesis, I would like to thank my supervisor Dr. Janusz Getta, for his constant support and encouragement throughout the course of this research.

Table of Contents

Chapter 1 Introduction

Chapter 2 Existing Solutions

Chapter 3 Performance Analysis

Chapter 4 Non-Regular Grids

Chapter 5 Analysis Of Results

Chapter 6 Conclusions

Appendix .

References .

List Of Figures

List Of Tables

CHAPTER 1

INTRODUCTION

Often there is a need to map multi dimensional space onto a one dimensional space. Mapping from a higher to a lower dimension is not unusual. As early as in our first programming course, we learnt how to take a two dimensional matrix and map it onto a linear range of memory addresses. A large number of non-standard database applications need to store and manipulate the representations of two-dimensional data objects. In this thesis, we focus on the problem of efficiently representing two-dimensional data objects in a one dimensional space, using the concept of linear clustering of disk areas that store information about the data objects.

1.1 Introduction

Two dimensional data objects consists of points, lines, rectangles, regions and surfaces. The representation of such data is becoming increasing important in applications as

computer-aided design, computer graphics, database management systems, computer vision, pattern recognition, solid modelling, robotics, and other areas [SEL87][GOO92]. Our goal in this thesis is to represent efficiently two dimensional data objects in a database, resulting in improved efficiency for accessing data from the database. A database consists of a collection of disk blocks, each being of equal size and containing several attributes or keys. A *query* is a request for all disk blocks that satisfy a predicate or have specific values or ranges of values for specified keys. Assuming that a database consists of N disk blocks with k keys apiece, there are two typical queries [SAM89]:

(a) A point query : it determines whether a given data object is in the database, and if so, yields the address corresponding to the block in which it is stored.

(b) A range query : (ie region search), which asks for a set of data objects whose specified keys have values within a given range.

Optimum representation of two dimensional data in a database is defined in terms of efficient execution of a query. A way of achieving efficient query execution is by maintaining *locality of data* . "Locality of data" means that data objects close to each other in two dimensional space are saved close to each other in the database. Maintaining locality of data in databases results minimum "access" time with a lesser amount of overhead in response to a query.

Many data structures currently used to represent two-dimensional data objects are hierarchical [BEC90][GUT84][ROB81]. They are based on the principle of recursive decomposition of space. Hierarchical data structures provide an efficient representation of two-dimensional data objects in one dimension, resulting in improved execution time. Hierarchical data structures are also attractive because of their conceptual clarity and ease of implementation. One such data structure is the *quadtree*

[SAM89][GAR82][SAM84][WEB85]. The resolution of the decomposition may be fixed beforehand, or it may be governed by the density of the input data objects.

Qaud-like decompositions are useful as space ordering methods [SAM89]. The purpose is to optimise the storage and processing sequences for two-dimensional data objects by mapping them into one dimension ie linearizing them. An efficient way to optimise storage and processing sequences is based on *linear clustering* of disk areas that store information about data objects residing in respective partitions. Linear clustering is exhibited when data objects close to each other in two-dimensional space are saved close together in one-dimensional space. A number of techniques for linear clustering of two-dimensional data objects have being proposed so far, including Gray curve [JAG90][FAL87], z-scan curve [ORE86], snake curve [JAG90], Hilbert curve [JAG90][BUT71] , etc.

The linear clustering approach attempts a decomposition of two-dimensional space using the concept of *grid-method*. In the grid-method, data objects are partitioned into small sub-sections by applying a grid on top of the two-dimensional space. Each sub-section is called a *grid-cell* [NIE84][BER91]. Each grid-cell encompasses a certain number of data objects which are part of the database. The linear mapping is performed in such a way that there is one-to-one mapping between the grid-cells and the disk blocks. The mapping should also preserve the spatial locality of the original two-dimensional image in one dimension.

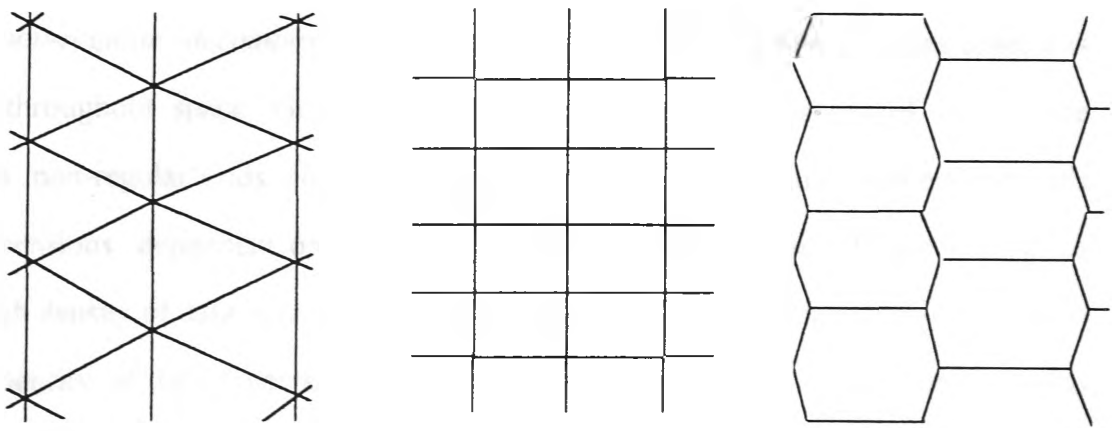
Space decomposition using the grid-method can be classified into two categories. The decomposition may be static or it may be governed by the density of the data objects in space. In the case of a space decomposition being static, called *regular decomposition,*

it is assumed that there is uniform density of data objects throughout the two-dimensional space. Grids used for regular decomposition of space are called as regular grids. In a regular grid, all grid-cell are of equal dimensions and each contains approximately similar amount of data objects. In general, any space decomposition scheme must possess the following properties [AHU81]:

(a) The partition should be an infinitely repetitive pattern in the plane . This would allow the representation to be useful for data objects of any sizes.

(b) The partition should be infinitely (recursively) decomposable into an increasingly fine pattern . This would allow the representation to be useful for data objects with arbitrarily fine spatial features.

Several schemes are used for recursive decomposition of two-dimensional space. Let us denote the number of sides of a face (grid-cell) in a given partition as 'm' and denote 'n' as the number of grid-cells meeting at a vertex . There exists only three partitions of the plane in which the value of m(n) is the same for all grid-cells, the possible (m,n) value being (4,4) , (3,6) and (6,3) [AHU81] Such tessellations correspond to a division of the plane into regular square, triangular and hexagonal cells (fig 1.1).



Triangular Tesselation Square Tessellation Hexagonal Tessellation

Fig 1.1 Types Of Tessellations.

All the three tessellations possess the first property demanded from a planar partition for image representation. With respect to the second property of recursive decomposition, the square and triangular tessellations possess this property (fig 1.2).

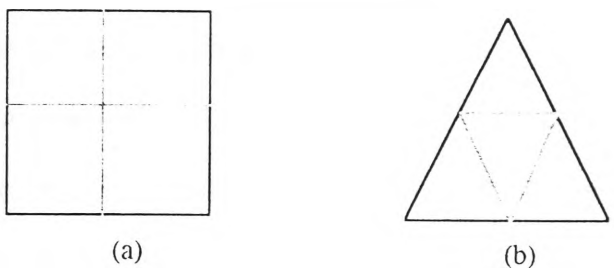


Fig 1.2 Cells (dotted lines) In (a) Square and (b) Triangular Tessellations : merge into larger cells (thick lines)

On the other hand, cells in a hexagonal tessellations cannot be further divided into regular congruent hexagons. Cells in tessellations involving only squares and triangles may be recursively partitioned into regular square and triangular decompositions.

In the case of space decomposition governed by the density of data objects in space, called a *non-regular decomposition*, it is not assumed that density of data objects is uniform throughout space. Grids used for non-regular decomposition of space are known as non-regular grids. Non-regular grids consists of rectangular shaped grid-cells with dimensions dependent on the density of data objects in space. Regions in space with a high density of data objects will have more grid-cells as compared to regions with a lower density of data objects, such that each grid-cell contains approximately equal number of data objects with in itself.

1.2 Research contributions

In this section, we present the goals of this thesis.

(i) *experimental analysis of space filling curves* : Applying a grid on top of two-dimensional space with uniform density data objects, data objects are partitioned into separate sub-regions called as grid-cells. The data objects in each grid-cell are saved onto the disk blocks in a database. There exists a one-to-one correspondence between the grid-cells and disk blocks. The mapping of grid-cells onto a database should preserve the spatial locality of the original two-dimensional image in one dimension. The result of the mapping is known as a *space-filling curve* [SAM84][JAG90]. A number of space filling curves exists, for example the Hilbert curve, snake-scan curve, z-scan curve and the Gray curve. One of the sub-goals of this thesis is to analyse the performance of space filling curves by conducting various experiments. On the basis of experimental results obtained, the space filling curves are arranged in order of their performance.

(ii) *algorithm for generation of a curve for a non-regular space decomposition* : In a non-regular decomposition of two-dimensional space, the grid-cells obtained are not of equal dimensions throughout the space, but depend on the density of data objects in space. The classical space filling curves are not applicable for non-regular grids as they require a uniform grid-cell size, to enable them to map the grid-cells onto the disk blocks. We propose an algorithm for mapping non-regular grid cells onto a database.

(iii) *analytical analysis of linear clustering curves* : The process of mapping grid-cells, obtained from regular or non-regular decomposition of space, onto one dimension is known as producing a *linear clustering curve*. Analysing the performance of linear clustering curves on the basis of experimental results is a tedious and time consuming job. An easier way of analysing performance is to test the performance of a curve

depending on its shape. We propose an algorithm by which we can analyse analytically the performance of a linear clustering curve.

1.3 Outline of the thesis

In this chapter we have presented an introduction to the concept of linear mapping of two dimensional data objects onto one dimensional storage space. In chapter two we present some linear mapping techniques for regular grids. We also propose a new linear mapping technique.

In chapter three we analyse experimentally the performance of the space filling curves. On the basis of the experimental results obtained we were able to arrange the space filling curves in order of their performance.

In chapter four we propose an algorithm for the generation of a curve useful in mapping non-regular grid cells onto a database. Two other trivial algorithms are also proposed for comparing the performance of the curves generated.

In the fifth chapter we analytically analyse the performance of the linear clustering curves. We propose an analytical algorithm by means of which we would be able to compare the performance of the curves. Finally, in chapter six we present a summary of this thesis along with avenues for further research.

CHAPTER 2

EXISTING SOLUTIONS

In the previous chapter we presented an overview of linearly representing two-dimensional data objects onto a one dimensional space using the concept of linear clustering. A number of techniques exists for linear clustering two-dimensional data objects, in this chapter we mention a few of them. We also propose a new linear clustering technique.

2.1 Linear mapping techniques

So far, we have discussed the regular and non-regular decomposition of two dimensional space, obtained on applying regular and non-regular grids respectively. A grid partitions the space into various sub-regions called *grid-cells*. We wish to optimise the storage and processing sequences for two-dimensional data by mapping the grid-cells lineally onto the database. A one-to-one mapping is maintained between the grid-cells and the disk blocks. The upper limit on the number of data objects a grid-cell can contain is determined by the disk block size.

A regular grid is useful in linearly representing two-dimensional data if there exists an uniform density of data objects in space. All regular grid cells are of equal dimensions, based on the disk block size. The one-to-one mapping of regular grid-cells onto the disk blocks is called as space filling curve.

In the case of non-uniform density of data objects in space, a non-regular grid is applicable. On applying a regular grid to space with non-uniform data density, we obtain an uneven distribution of data in the database. Some disk blocks may have an overflow of data while some may be totally empty. In a non-regular grid, grid-cell size is dependent on density of data objects in space. Space filling curves cannot be applied to non-regular grids as they require grid-cells to be of equal dimensions. Later on in this thesis, we propose an algorithm for mapping the non-regular grid cells onto the database.

In the following section we present a number of space filling curves and examine their different characteristics.

2.1.1 Space filling curves

A brief description of various space filling curves is given below. We first consider the space filling curves with respect to grids with square tessellations. We have made the assumption that the regular grid is of equal dimensions in the x-axis and y-axis direction, which is equal to a power of two.

2.1.1.1 Snake Curve

This is the simplest mapping function [JAG90]. The position of a grid cell is dependent on the sequential position of a grid-cell on the snake-scan access line.

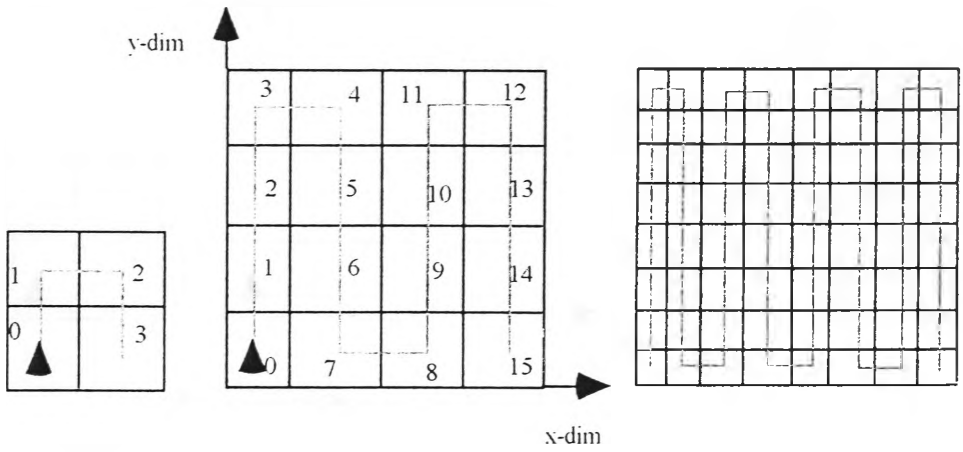


Fig 2.1 Snake Curve

We present an algorithm for calculating sequence in which a grid-cell is accessed by the snake curve. Each grid-cell is identified by the value of its top left co-ordinate (x,y).

Algorithm.1: (Snake Curve)

```
(1)Begin
(2)     input : (x,y) value of a grid-cell co-ordinate.
        input : (x_dim,y_dim) dimensions of the grid.
(3)     if (even(x))
(4)             return (x * (y_dim+1) + y)
(5)     else
(6)             return ((x+1)* (y_dim + 1) - y - 1)
(7)End
```

2.1.1.2 Column-wise Curve

The column-wise curve is nearly similar in shape to the snake curve [JAG90]. Position of a grid-cell in database is dependent on the sequential position on the grid-cell on the column-wise curve access line.

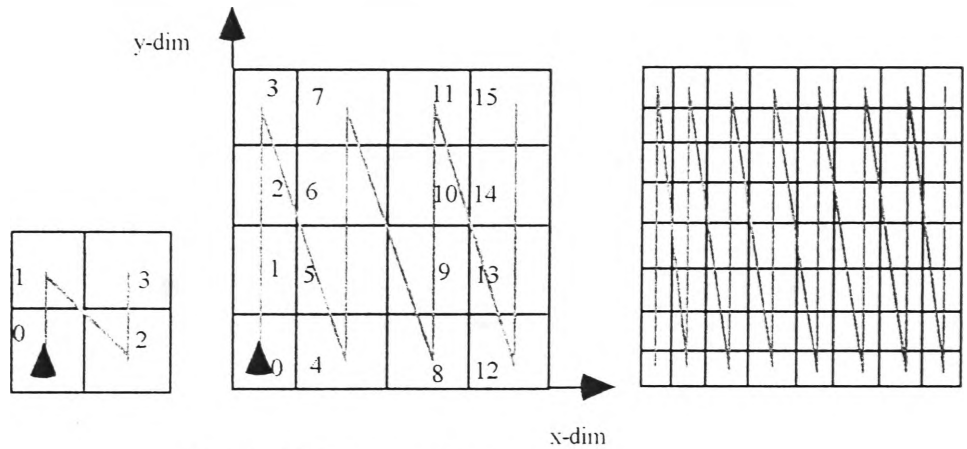


Fig 2.2 Column-wise Curve

We present an algorithm for calculating sequence in which a grid-cell is accessed by the column-wise curve. Each grid-cell is identified by the value of its top left co-ordinate (x,y).

Algorithm.1: (Column-wise Curve)
(1)**Begin**
(2) input : (x,y) value of a grid-cell co-ordinate.
input : (x_dim,y_dim) dimensions of the grid.
return (x* (y_dim + 1) + y)
(7)**End**

2.1.1.3 Hilbert Curve

The Hilbert curve was proposed by Faloutsos and Roseman [SAM89][BUT71]. The shape of the curve is based on the Peano curve. In a Hilbert curve two nearest neighbours

are usually mapped to a point not far away in the linear traversal, thus having a slightly higher degree of locality as compared to other curves. The Hilbert curve exhausts a quadrant or sub-quadrant of a square image before exiting it.

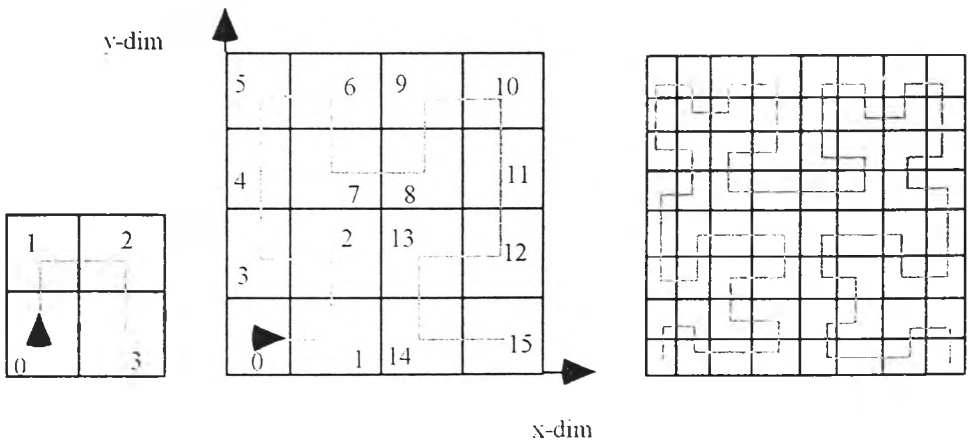


Fig 2.3 Hilbert Curve

We present an algorithm for calculating sequence in which a grid-cell is accessed by the Hilbert curve. The algorithm consists of two functions, namely quadrant_number() and hilbert(). The quadrant_number() function takes as input the (x,y) values of a point and returns the quadrant number of a grid in which the point exists. The quadrants are numbered from 0 to 3, starting from the lower left quadrant and travelling clockwise. The hilbert() is a recursive function which returns the linear position of a grid-cell. Both functions also require the dimension of the grid, n, among their input parameters.

Algorithm 3: (Hilbert Curve)

```
(1)Begin
(2)hilbert( x , y , n)
(3)begin
(4)if ((n mod 2) = 0) /*if grid dimension of power 2*/
(5)begin
(6)     m = quadrant_number( x , y , n).
(7)
(8)     if (n _ 2)   /*if grid size greater than 2*/
(9)     begin
(10)          if (m = 1) OR (m = 2)
(11)          begin
```

```
(12)            new_x = x mod (n div 2).
(13)            new_y = y mod (n div 2).
(14)        end
(15)        if (m = 0)
(16)        begin
(17)            new_y = x mod (n div 2).
(18)            new_x = y mod (n div 2).
(19)        end
(20)        if (m = 3)
(21)        begin
(22)            new_x = (n div 2) - y mod (n div 2) - 1.
(23)            new_y = (n div 2) - x mod (n div 2) - 1.
(24)        end
(25)        x = new_x.
(26)        y = new_y.
(27)        return( m * (n div 2) * (n div 2) + Hilbert( x , y , (n div 2) ).
(28)  end
(29)  else  /*if grid size equal to 2*/
(30)        return(m).
(31)end
(32)end
(33)
(34)  quadrant_number( x , y , n)
(35)  begin
(36)  if (x < (n div 2)) AND (y < (n div 2))
(37)        return 0.
(38)  if (x < (n div 2)) AND (y >= (n div 2))
(39)        return 1.
(40)  if (x >= (n div 2)) AND (y >= (n div 2))
(41)        return 2.
(42)  if (x >= (n div 2)) AND (y < (n div 2))
(43)        return 3.
(44)  end
(45)END.
```

2.1.1.4 Z-Scan Curve

The z-scan curve was proposed by Orenstein [ORE86]. The curve is recursive, in that it consists of "N" shape (covering four points) repeated throughout the space. Groups of

four Ns are connected in a N pattern; groups of four of these groups of four are connected in the same pattern, etc.

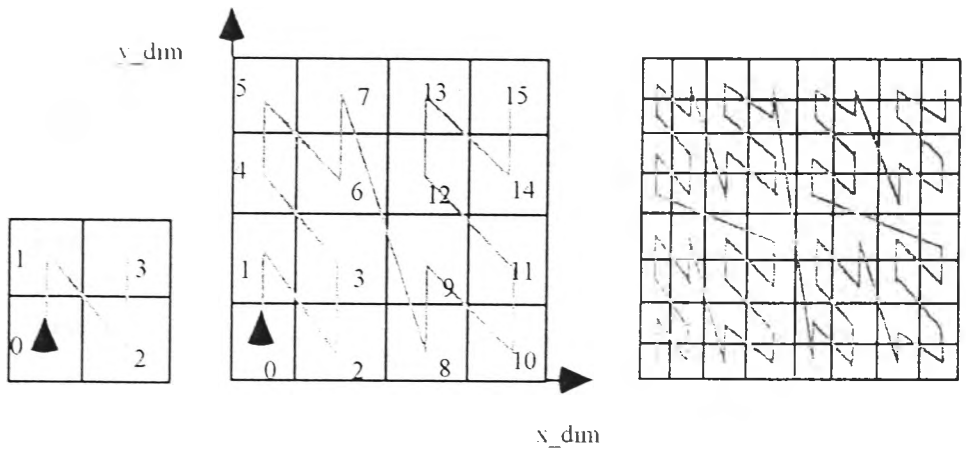


Fig 2.4 Z-Scan Curve

We present an algorithm for calculating the sequence in which grid-cell is accessed by the z-scan curve. A grid cell is identified by the value of its top left co-ordinate (x,y). The algorithm consists of two functions, namely quadrant_number() and zscan(). The quadrant_number() function takes as input the (x,y) values of a point and returns the quadrant number of a grid in which the point exists. The quadrants are numbered from 0 to 3, starting from the lower left quadrant and travelling clockwise. The zscan() function returns the linear position of a grid-cell. Both functions also require the dimension of the grid, n, among their input parameters.

Algorithm 4 : (Z-Scan Curve).

```
(1)Begin
(2)zscan (x , y , n)
(3)begin
(4)if ((n mod 2) = 0) /*if grid dimension of power 2*/
(5)begin
(6)sum = 0
(7)      while (n _ 2)
(8)       begin
(9)           m = quadrant_number( x , y , n).
(10)          if (m = 0)
```

```
(11)        begin
(12)              new_x = x.
(13)              new_y = y.
(14)        end
(15)        if (m = 1)
(16)        begin
(17)              new_x = x.
(18)              new_y =  y - (n div 2).
(19)        end
(20)        if (m = 2)
(21)        begin
(22)              new_x = x - (n div 2).
(23)              new_y = y.
(23)        end
(24)        if (m = 3)
(25)        begin
(26)              new_x = x - (n div 2).
(27)              new_y = y - (n div 2).
(28)        end
(29)        x =  new_x.
(30)        y = new_y.
(31)        sum = sum + (m * (n div 2) * (n div 2).
(32)        n = n div 2.
(33)     end
(34)       if ( n = 2)
(35)        begin
(36)          m = quadrant_number(x , y , n).
(37)          sum = sum + m.
(38)         end
(39)   return(sum).
(40)end
(41)end
(42)
(43)quadrant_number(x , y, n)
(44)   begin
(45)   if (x < (n div 2)) AND (y < (n div 2))
(46)          return 0.
(47)   if (x < (n div 2)) AND (y >= (n div 2))
(48)          return 1.
(49)   if (x >= (n div 2)) AND (y >= (n div 2))
(50)          return 2.
(51)   if (x >= (n div 2)) AND (y < (n div 2))
(52)          return 3.
(53)    end
(54)END.
```

2.1.1.5 Gray Curve

This curve was proposed by Faloutsos [JAG90][FAL87]. The Gray curve is based on the (binary) Gray Code, in which numbers are coded into binary representations such that successive numbers differ in exactly one bit position. Faloutsos observed that difference in only one bit position had a relationship with locality and proposed that points in two-dimensional spacce can be gray coded to obtain their one-dimensional co-ordinate.

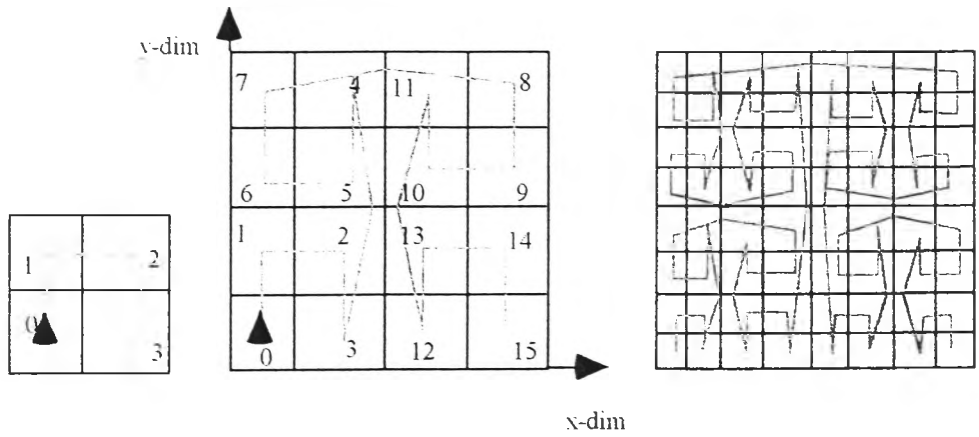


Fig 2.5 Gray Curve

We present an algorithm for calculating the sequence in which a grid-cell is accessed by the Gray curve. A grid cell is identified by the value of its top left co-ordinate (x,y). The algorithm consists of two functions, namely quadrant_number() and gray(). The quadrant_number() function takes as input the (x,y) values of a point and returns the quadrant number of a grid in which the point exists. The quadrants are numbered from 0 to 3, starting from the lower left quadrant and travelling clockwise. The function gray() is a recursive function which returns the linear position of a grid-cell. Both functions also require the dimension of the grid, n, among their input parameters.

Algorithm 5: (Gray Curve)

```
(1)Begin
(2)gray( x , y , n)
(3)begin
(4)if ((n mod 2) = 0) /*if grid dimension is of power 2*/
(5)begin
(5)     m = quadrant_number( x , y , n).
(6)
(7)     if (n _ 2)
(8)     begin
(9)
(10)          if (m = 0)  OR (m = 3)
(11)          begin
(12)               new_x = x mod (n div 2).
(13)               new_y = y mod (n div 2).
(14)          end
(15)          if (m = 1)  OR (m = 2)
(16)          begin
(17)               new_x = (n div 2) - 1 - ( x mod (n mod 2)).
(18)               new_y = (n div 2) - 1 - (y mod (n mod 2)).
(19)          end
(20)          x = new_x.
(21)          y = new_y.
(22)          return( m * (n div 2) * (n div 2) + gray( x , y , (n div 2) ).
(23)    end
(24)    else   /*if grid size equal to 2*/
(25)          return(m).
(26)    end
(27)end
(28)
(29)    quadrant_number( x , y , n)
(30)    begin
(31)    if (x < (n div 2)) AND (y < (n div 2))
(32)          return 0.
(33)    if (x < (n div 2)) AND (y >= (n div 2))
(34)          return 1.
(35)    if (x >= (n div 2)) AND (y >= (n div 2))
(36)          return 2.
(37)    if (x >= (n div 2)) AND (y < (n div 2))
(38)          return 3.
(39)    end
(40)END.
```

2.1.1.6 Cantor-diagonal Curve

The Cantor-diagonal curve starts from the rightmost corner of a grid and access grid-cells in a diagonal order [SAM89]

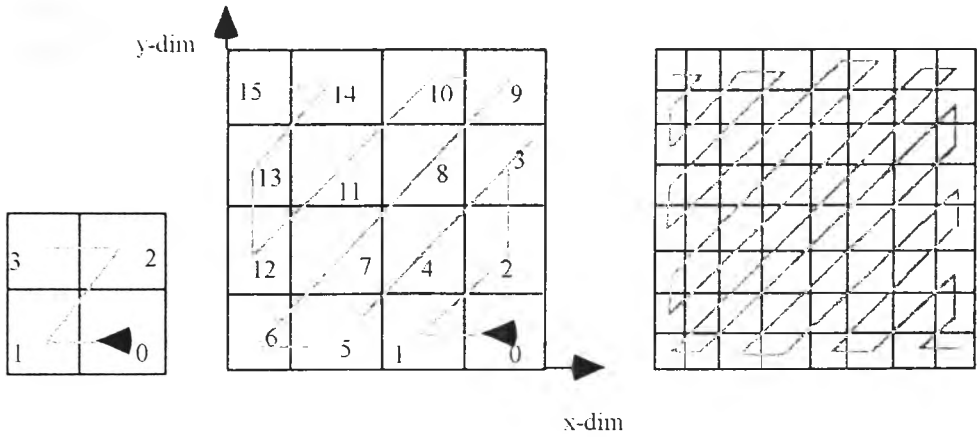


Fig 2.6 Cantor-diagonal Curve

We present an algorithm for calculating the position of a grid-cell with respect to Cantor-diagonal curves access line. A grid cell is identified by the value of its top left co-ordinate (x,y). The algorithm takes as input the (x,y) values of a point and returns the linear position of a grid-cell. The algorithm also requires the dimension of the grid, n, among its input parameters.

Algorithm 6: (Cantor-diagonal Curve)

```
(1)Begin
(2) count = 0, temp_x = n, temp_y = 0.
(4)Cantor-diagonal( x , y , n)
(5)begin
(6)     if (temp_x = x) and (temp_y = y)
(7)             return count;
(8)     while (temp_x  _ x) and (temp_y  _ y)
(9)     begin
(10)            if ( temp_x  _ 0)
(11)            begin
(12)                    y = y + 1.
(13)                    count = count + 1.
```

```
(14)        end
(15)        else
(16)        begin
(17)              x = x - 1.
(18)              count = count + 1.
(19)        end
(20)        while (x _ n)
(21)        begin
(22)              x = x + 1.
(23)              y = y + 1.
(24)              count = count + 1.
(25)              if (temp_x = x) and (temp_y = y)
(26)                    return count;
(27)        end
(28)        if ( temp_y _ n)
(29)        begin
(30)              x = x - 1.
(31)              count = count + 1.
(32)        end
(33)        else
(34)        begin
(35)              y = y - 1.
(36)              count = count + 1.
(37)        end
(38)        while (y _ 0)
(39)        begin
(40)              x = x - 1.
(41)              y = y - 1.
(42)              count = count + 1.
(43)              if (temp_x = x) and (temp_y = y)
(44)                    return count;
(45)        end
(46)  end
(47)end
(48)END.
```

2.1.1.7 Spiral Curve

The Spiral curve starts from the centre of the grid and then accesses grid cells in a spiral order. The spiral curve is attractive when ordering a space which is unbounded in the four directions emanating from the origin.

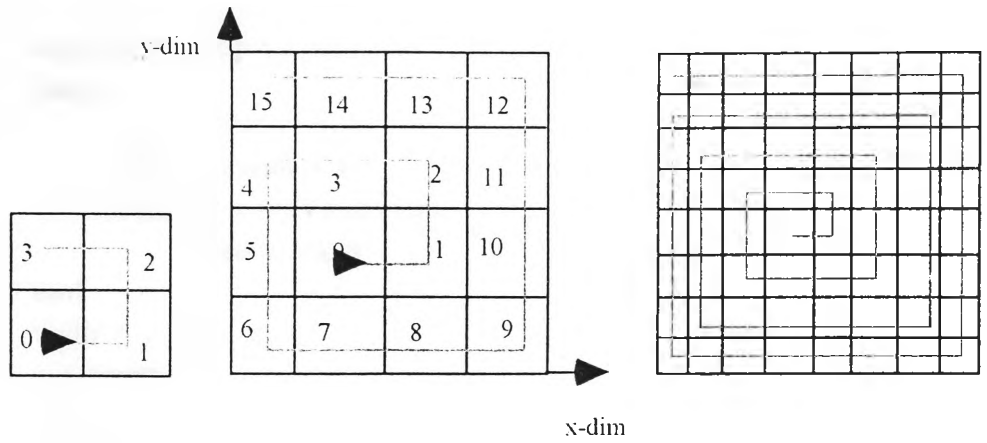


Fig 2.7 Spiral Curve

We present an algorithm for calculating the position of a grid-cell using a spiral curve access line. A grid-cell is identified by the value of its top left co-ordinate (x,y). The algorithm takes as input the (x,y) values of a point and returns the linear position of a grid-cell. The algorithm also requires the dimension of the grid, n, among its input parameters.

Algorithm 7: (Spiral Curve)

```
(1)Begin
(2) count = 0, temp_x = (n div 2), temp_y = (n div 2).
 (3) x_dir = 0, y_dir = 0.
(4)spiral( x , y , n)
(5)begin
(6)     if (temp_x = x) and (temp_y = y)
(7)             return count;
(8)     while (temp_x  _ x) and (temp_y  _ y)
(9)     begin
(10)            cnt_temp = 0.
(11)            while (cnt_temp ² x_dir)
(12)            begin
(13)                    x = x + 1.
(14)                    count = count + 1.
(15)                    if (temp_x = x) and (temp_y = y)
(16)                            return count;
(17)            end
(18)            x_dir = x_dir + 1.
(19)            cnt_temp = 0.
```

```
(20)        while (cnt_temp < y_dir)
(21)        begin
(22)              y = y + 1.
(23)              count = count + 1.
(24)              if (temp_x = x) and (temp_y = y)
(25)                    return count;
(26)        end
(27)        y_dir = y_dir + 1.
(28)        cnt_temp = 0.
(29)        while (cnt_temp < x_dir)
(30)        begin
(31)              x = x - 1.
(32)              count = count + 1.
(33)              if (temp_x = x) and (temp_y = y)
(34)                    return count;
(35)        end
(36)        x_dir = x_dir + 1.
(37)        cnt_temp = 0.
(38)        while (cnt_temp < y_dir)
(39)        begin
(40)              y = y + 1.
(41)              count = count + 1.
(42)              if (temp_x = x) and (temp_y = y)
(43)                    return count;
(44)        end
(45)        y_dir = y_dir + 1.
(46)        cnt_temp = 0.
(47)  end.
(48)end.
(49)END.
```

2.1.2 Applying curves to grids with non-square grid-cells

In this section, we discuss space filling curves with respect to grids consisting of triangular or hexagonal shaped grid-cells. Figure 2.8 shows a relative correspondence of grid-cells between rectangular and non-rectangular grids. In hexagonal grids, odd columns are positioned at a lower level than columns, so that rows in a hexagonal grid are

arranged in a zig-zag manner. In the case of triangular grids, neighbouring grid-cells of a grid-cell are inverted in shape. Rows in a triangular grid consists of a sequence of alternating upright and inverted triangles.

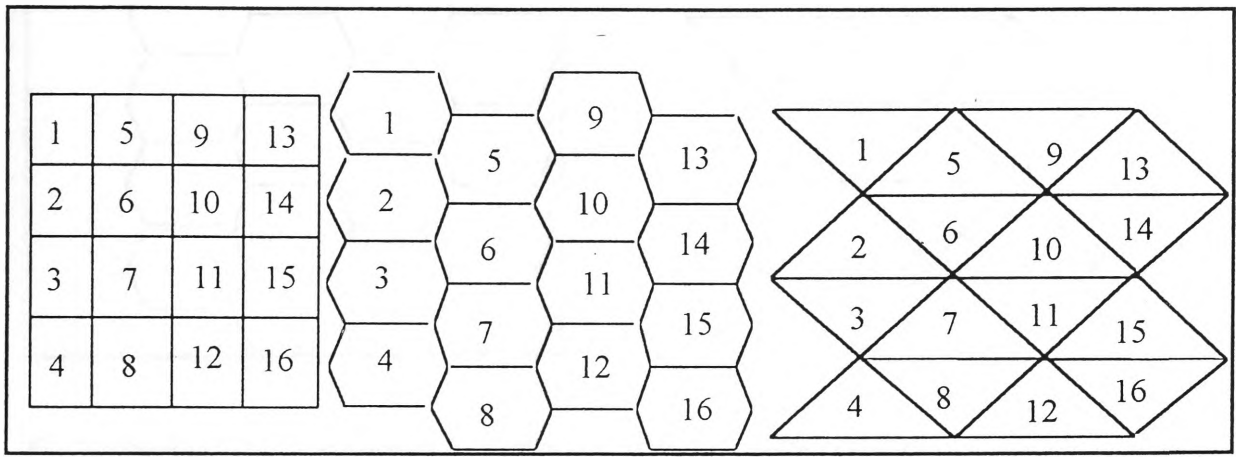


Fig 2.8 Corresponding grid-cells in rectangular and non-rectangular grids

Square, triangular and hexagonal shaped grid-cells are of equal area with respect to each other. While the square grid-cells are identified by the value of their top left co-ordinate, triangular and hexagonal grid-cells are identified by their centroid values. Depending on the value of the co-ordinates of a triangular or hexagonal grid-cell, the centre point is calculated which identifies a grid-cell. We can apply the same algorithms discussed in the previous section for different space filling curves, in order to find the position of a hexagonal or triangular grid-cell in database.

Below we present the mapping of different space filling curves for grids made up of triangular or hexagonal shaped grid-cells.

*Snake Curve:

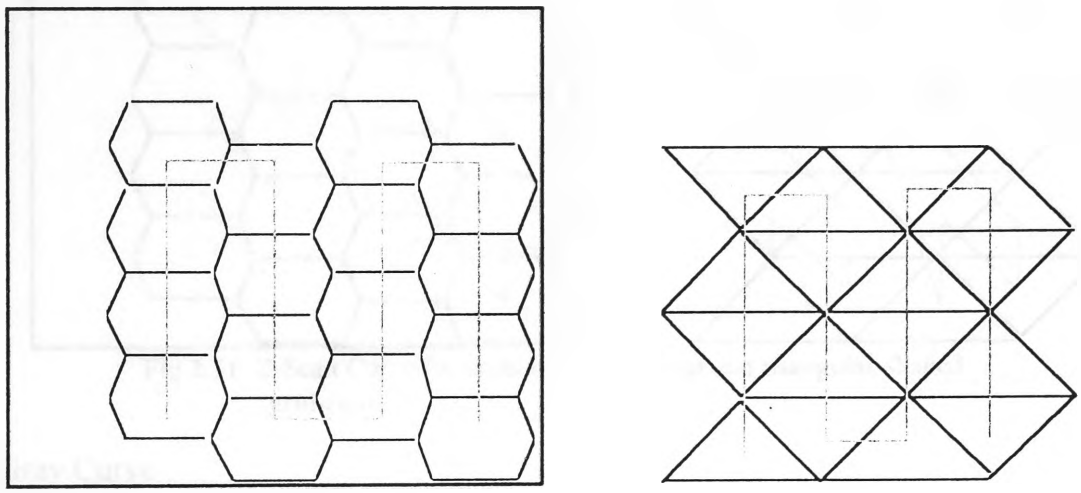


Fig 2.9 Snake Curve for grids with hexagonal and triangular shaped grid cells

* Hilbert Curve:

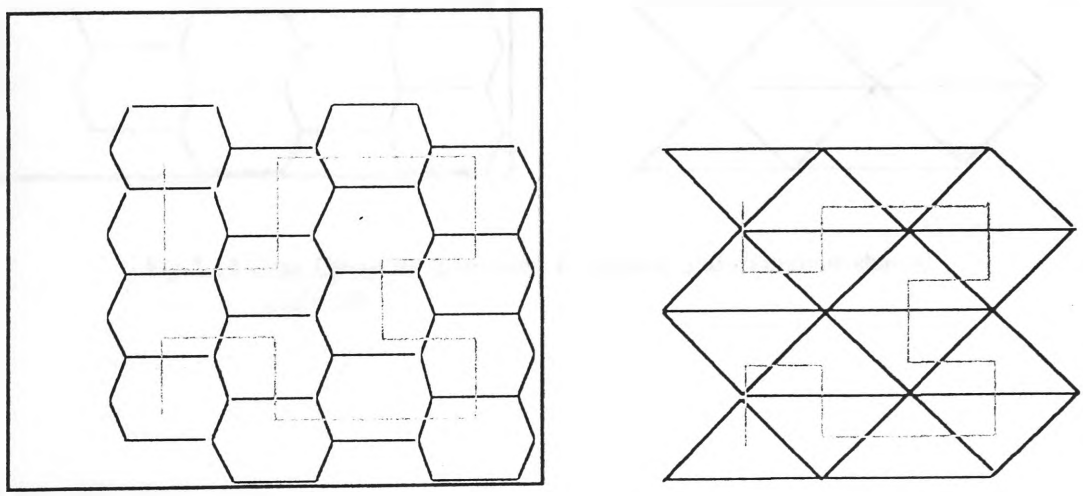


Fig 2.10 Hilbert Curve for grids with hexagonal and triangular shaped grid-cells

* Z-Scan Curve :

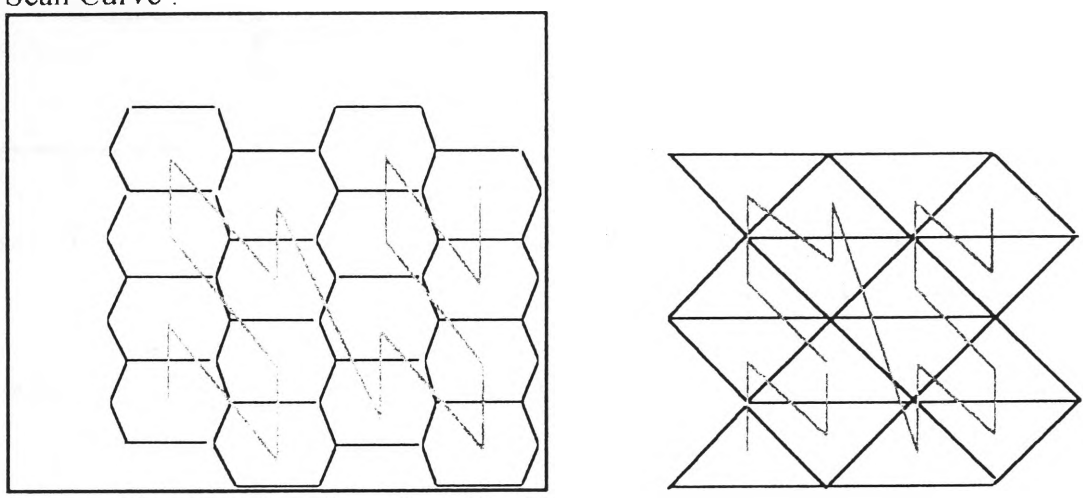


Fig 2.11 Z-Scan Curve for grids with hexagonal and triangular shaped grid-cells

* Gray Curve :

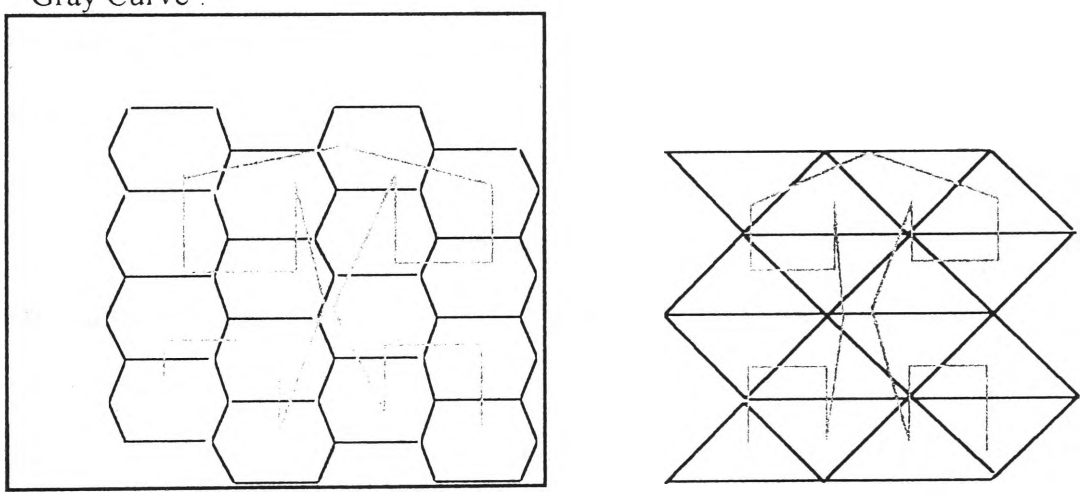


Fig 2.12 Gray Curve for grids with hexagonal and triangular shaped grid-cells

2.2 The H-scan curve

In this section, we propose a new space filling curve . As the shape of the curve is similar to the alphabet "H" , we have named it the "H-Scan Curve". The curve shape is shown in figure-2.13.

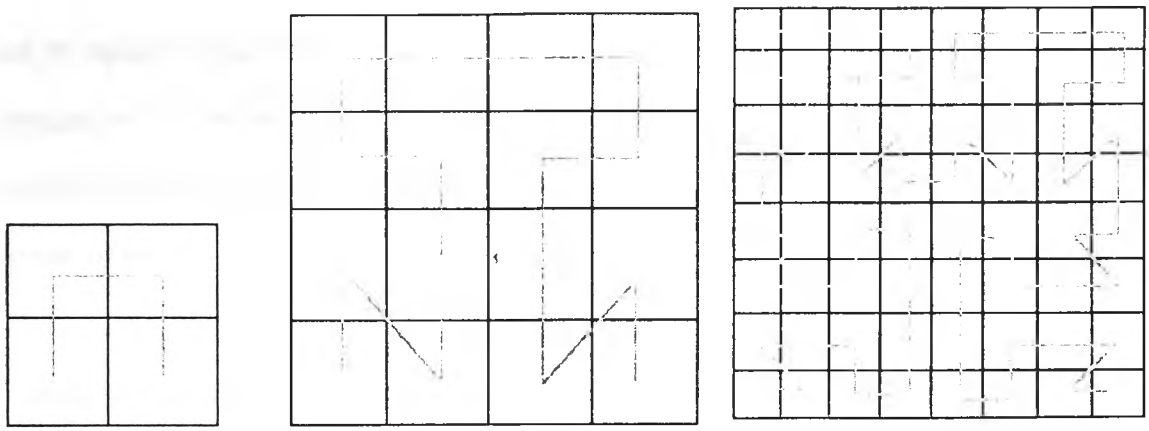


Fig 2.13 H-scan Curve

In the H-scan curve, we try to combine the main features of the Hilbert curve, the snake curve and the z-scan curve. A grid cell is mapped onto a database record in a sequence in which it is accessed by the H-scan curve. H-scan curve ordering is a total ordering of grid cells, this property leads to the observation that the only possible relationship between grid cells are containment and precedence (in H-scan curve order). Overlap cannot occur . Much of the usefulness of the H-scan curve mapping is based on the property that it preserves proximity, ie. if two points are close together in space they will also be close together in the H-scan curve ordering. By preserving proximity , data are clustered for efficient access on secondary storage.

2.2.1 Analysis of the H-scan curve

In this section, we will discuss some features of the H-scan curve.

- Space Requirements: In figure 2.13, we mapped the H-scan curve for a grid with dimensions 2X2, 4X4 and 8X8 units. Decomposition of two dimensional rectangular area of size U x V, where the lower left corner is at (0,0), is performed so that the grid-cells

are of equal dimensions. The grid-cell size is directly proportional to the density of data objects in two dimensional space. The greater the density of data in space, the more the number of grid-cells. The number of grid-cells along the U and V direction are the same power of two.

- Integration with the database: The grid-cells are saved onto the database blocks in the sequence in which the H-scan curve accesses the grid points. Each grid-cell is mapped onto a unique database record.

- Proximity: Proximity is maintained in the H-scan curve. It is not possible to map an m-dimensional grid into an n-dimensional grid, for any different n and m in such a way that two points that are close together in the former are always close together in the latter. Considering nearest neighbours along the grid axes only, we can see that each grid point has 2m nearest neighbours in the former, and only 2n nearest neighbours in the latter. So at least 2(n - m) points are not neighbours in n - dimension as in m - dimension.The best we could hope to achieve in two-dimensional is to have two of the four nearest neighbours.

- Performance: Later on in this thesis, we analyse the H-scan analytically and also on basis of experimentally results. The H-scan curve gives a good performance as compared with other space filling curves, it performs similarly to a Hilbert curve which provides us with the best performance.

2.3 Summary

In this chapter, we presented a brief overview of some existing space filling curves and proposed a new space filling curve called the H-scan curve.

In the next chapter, we experimentally analyse the performance of space filling curves and arrange them in order of performance.

CHAPTER 3

PERFORMANCE ANALYSIS

So far, we have discussed some existing space filling curves and also presented a new space-filling curve called the H-scan curve. In this chapter, we will analyse the space filling curves on basis of experimental results and arrange them in increasing order of performance.

3.1 Experiments for analysing curve performance

In this section, we present a brief outline of the experiments performed for analysing performance of a space filing curve and also mention the type of queries used.

The experiments performed for analysing performance of the space filling curves can be divided into two groups:

GROUP A: In the previous chapter, we had applied space-filling curves to grids made up of square, triangular and hexagonal shaped grid-cells. Our aim is to check if the

performance of a space filling curve is dependent on shape of a grid cell. We experimentally evaluated the performance of a space-filling curve for each of the three types of grids and compared the results obtained.

GROUP B: In the second group, we conducted experiments to select the curve which provides the most efficient performance among all the space filling curves. On basis of the experimental results we were able to arrange the space-filling curves in order of their performance.

For all the experiments conducted only range queries were considered. Range queries, called "search windows", in two dimensional space specify a rectangular area overlapping some grid-cells in a grid. Each grid-cell in a grid is identified by its co-ordinate values specifying the top left corner and bottom right corner of the grid-cell respectively. A specified search window will overlap with one or more grid cells. A search window is specified by its top left co-ordinate and bottom-right co-ordinate, (x1,y1) and (x2,y2) respectively. In response to the search window query, all grid cells are fetched whose co-ordinate values oerlap the search window.

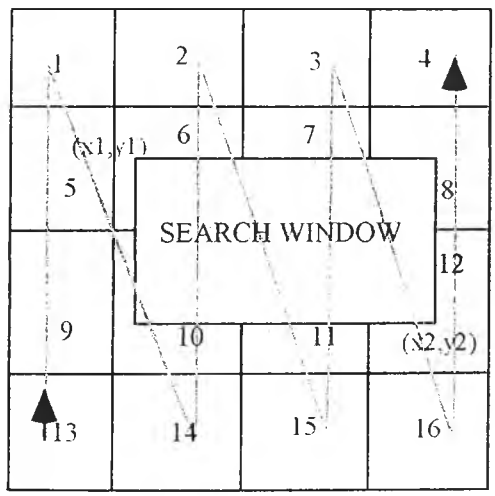


Fig 3.1 Search Window

In figure 3.1, a search window is specified on a square regular grid in which grid-cells are traversed with respect to a column-wise curve access line. We will have to access data in grid cells 6, 7, 8, 10, 11 and 12 in response to the search window query specified on the grid.

We have considered the following three search window shapes :

- Horizontal search window : A search window which is landscape oriented, length of the side parallel to the x-axis is longer as compared to length of the rectangle side parallel to the y-axis.

- Vertical search window : A search window which is portrait oriented, length of side parallel to the x-axis is smaller as compared to the length of the rectangle side parallel to the y-axis.

- Square search window : Length of rectangle side parallel to the x-axis and parallel to the y-axis is same.

3.1.1 GROUP A: Analysing performance of space-filling curves with respect to different grid-cell shapes.

In the previous chapter, we applied space-filling curves to grids made up of square, triangular and hexagonal shaped grid-cells. We assumed that all grid cells were of equal area and that the dimensions of the grid in x-axis and y-axis direction was same, power of two. In this section, we analyse experimentally the performance of the space-filling curves when applied to grids with the three type of tessellations. The grid considered was 16X16 units. On specifying a search window query, the measure of performance for a space filling curve was the total number of grid-cells traversed by the curve in order to access all the grid-cells overlapped by the search window. The search window shapes

considered were square, horizontal and vertical. Position and size of the search window were calculated randomly in proportion to the dimensions of the grid, rejecting a search window if it extends outside the grid.The results presented below in mean and standard deviation form are obtained from ten trials of hundred different search windows.

	SQUARE GRIDS	TRIANGULAR GRIDS	HEXAGONAL GRIDS
HILBERT CURVE	5.19±1.23	5.45±0.11	5.50±1.22
Z-SCAN CURVE	7.80±1.19	8.02±1.01	7.77±1.44
GRAY CURVE	7.82±1.92	7.92±0.18	7.88±2.45
SNAKE SCAN CURVE	6.54±0.88	6.81±0.44	6.52±1.59

Table 3.1 Experimental results obtained on applying the space-filling curves to grids with different grid-cell shapes

3.1.1.1. Analysis

From the table presented above, we can infer that a change in grid-cell shape does not affect the performance of a space filling curves. The experimental readings obtained for each space-filling curve were approximately similar to one another. As the performance of the space filling curves does not depend on the shape of grid-cells, in our forth coming experiments we only use grids with square shaped grid cells.

3.1.2 GROUP B: Experiments for selecting the best space filling curve

We conducted a number of experiments on the space filling curves for analysing their performance. The experiments performed can be divided into three major sub-divisions, each consisting of four sub-experiments. Experiments in each of the three sub-divisions were conducted under different conditions and parameters.

The major sub-divisions are:

(i) Sub-division 1: For experiments conducted under this sub-division, dimensions of the grid taken into consideration were 128 X 128 units and each grid cell was of 1X1 units ie. there where 128 grid cells parallel to the x-axis direction and 128 grid cells parallel to the y-axis direction. The search window shapes considered were square , horizontal and vertical. Position and size of the search window were calculated randomly, in proportion to the dimension of the grid. The maximum dimension of the search window was 64X64 units and the minimum was 2X2 units. The results presented in this sub-division are mean values of ten trials of a thousand readings each.

(ii) Sub-division 2: The grid dimensions taken into consideration for experiments conducted under this sub-division were 8X8, 16X16, 32X32, 64X64 and 128X128 units. Dimension of each grid cell was 1X1 units. Position and size of a search window were calculated randomly in proportion to the grid size, for eg, in a grid of 64X64 units the search window will have a maximum size of 32X32 units.

(iii) Sub-division 3: In this sub-division, we kept the size of the grid constant while varying the search window. The position and size of the search window were calculated randomly. Grid dimensions considered were 128X128 units and cell dimension was 1X1

unit. The area of the search windows lay between the range of 0-16, 16-32, 32-48, 48-64 and 64-96 units.

Four common experiments were conducted in each of the three above mentioned sub-divisions. The experiments are :

(i) Sequential Block Count : On specifying a search window query on two dimensional data, it will overlap a number of grid-cells. In this experiment, we count the maximum number of grid-cells a space-filling curve can access in sequence which lie inside the search window, without accessing an outside grid-cell. By testing for sequentially arranged grid-cells, we in turn measure how close together neighbouring two-dimensional data is saved onto a database.

(ii) Number of "hops" : A "hop" occurs when the curve accesses a grid-cell not overlapped by the search window from a grid-cell inside the search window. We count the number of times a hop is taken by a space-filling curve. By counting the number of hops, we measure how many "reads" are to be made to access data from a database.

(iii) Linear Span : In measuring linear span of a curve we count the total number of grid-cells accessed, overlapped or not-overlapped by a search window, to read grid-cells overlapped by a search window. By linear span of a curve we measure the total number of disk blocks accessed in response to a query.

(iv) Fraction of Blocks Read Sequentially : In this experiment, we measure the mean number of grid-cells accessed sequentially. We calculate the mean number of the grid-cells arranged sequentially within the range of a search window as compared to the total

number of grid-cells accessed. By measuring the fraction of cells read sequentially, we measure the average number of data records arranged sequentially as compared to the linear span.

In the next three sub-sections, we present experimental results obtained for analysing the performance of the space filling curves.

3.1.2.1 Sub-division 1 :

The results presented in this sub-section are mean values of ten trials of a thousand readings.

1. Sequential Block Count :

Space Filling Curve	Number of Blocks Accessed Sequentially
Hilbert Curve	228 ± 4
Gray Curve	160 ± 6
Z-Scan Curve	150 ± 8
Snake Curve	40 ± 4
H-Scan Curve	169 ± 6

Table 3.2 Sequential Block Count In Exp.1

Analysis : From the above table we can infer that the Hilbert curve gives the best performance while the snake curve provides us with the worst performance. As the snake

curve access grid-cells in a column-by-column fashion, the maximum grid-cells it can access sequentially, when a search window is specified, is dependent on the height of the search window. The Gray curve, Z-scan curve and H-scan curve provides us with approximately similar results, which are better than the snake curve but not as good as the performance of the Hilbert curve.

2. Total Number of Hops :

Space Filling Curve	Total Number of Hops
Hilbert Curve	26 ± 3
Gray Curve	44 ± 8
Z-Scan Curve	44 ± 10
Snake Curve	29 ± 6
H-Scan Curve	41 ± 7

Table 3.3 Number Of Hops In Exp.1

Analysis : The lesser the number of hops taken by a curve, the better its performance. The Hilbert curve again provides us with the best performance while the snake scan curve comes a close second. The Z-scan curve and Gray curve gives the worst performance in this case. The reason for their bad performance is the large number of 'jumps' the curves take while accessing grid-cells.

3. Linear Span :

Space Filling Curve	Total number of Grid-Cells Accessed
Hilbert Curve	4780 ± 8
Gray Curve	5412 ± 12
Z-Scan Curve	3514 ± 11
Snake Curve	3213 ± 7
H-Scan Curve	4233 ± 11

Table 3.4 Linear Span In Exp.1

Analysis : In this experiment, the lesser number of grid-cells accessed by a space-filling curve the better its performance. In this case the snake curve gives the best performance along with the Z-scan curve .

4. Fraction of Blocks Read Sequentially

Space Filling Curve	Fraction of Blocks Read Sequentially
Hilbert Curve	0.89 ± 0.03
Gray Curve	0.75 ± 0.05
Z-Scan Curve	0.77 ± 0.09
Snake Curve	0.81 ± 0.04
H-Scan Curve	0.74 ± 0.04

Fig 3.5 Mean Number Of Blocks Accessed Sequentially In Exp.1

Analysis : On average, all the curves perform well. The snake scan curve and the Hilbert curve provide us with similar results. It can be inferred that the mean number of grid-cells accessed sequentially is directly proportional to the number of hops taken by a curve while accessing grid-cells within a search window

3.1.2.2 Sub-division 2 :

For experiments conducted under this sub-section, the grid dimensions taken into consideration were 8 X 8 , 16 X 16 , 32 X 32 , 64 X 64 and 128 X 128 units .Results presented below are the mean values of thousand readings .

In the graphs below we have used the following notations:

NOTATION	CURVE
H	HILBERT CURVE
S	SNAKE CURVE
C	H-SCAN CURVE
Z	Z-SCAN CURVE
G	GRAY CURVE

Table 3.6 Graphical notations

1. Sequential Block Count :

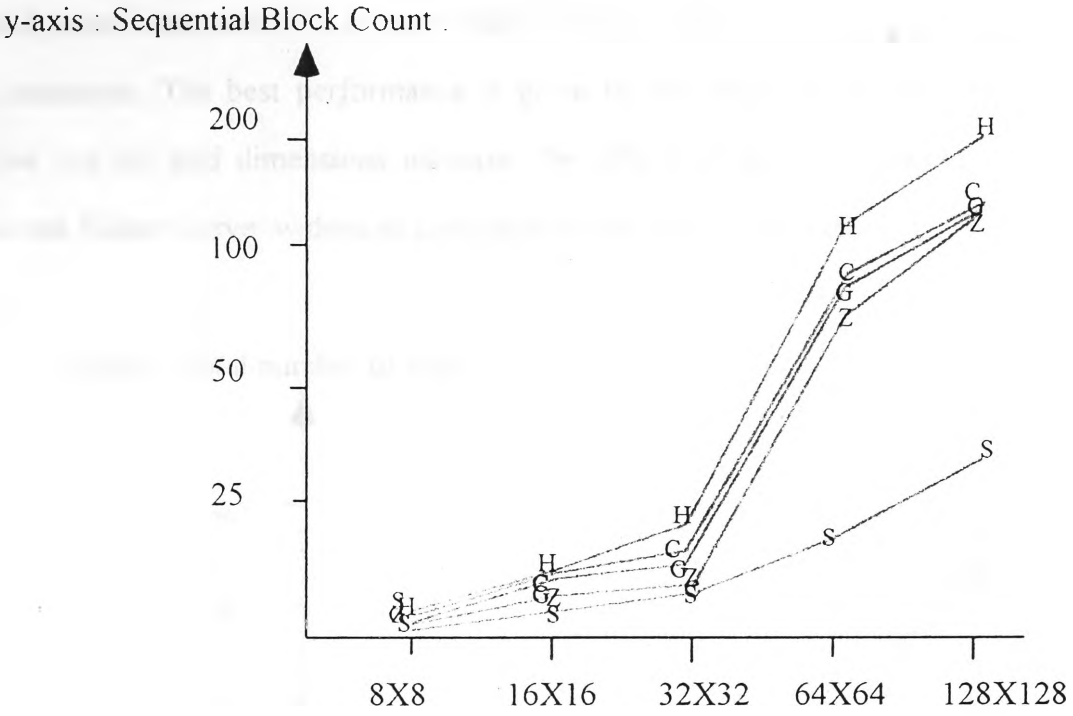


Fig 3.2 Sequential Blocks Accessed in Exp.2

Curve	8X8	16X16	32X32	64X64	128X128
H	±1	±2	±2	±5	±4
S	±2	±2	±3	±4	±6
C	±1	±2	±3	±6	±8
Z	±2	±3	±2	±5	±4
G	±2	±4	±3	±6	±6

Table 3.7 Error Table for Fig. 3.2

Analysis : In grids with smaller dimensions the number of grid-cells accessed sequentially by all the space filling curves is approximately similar. As the grid dimensions increase, the performance of the snake curve begins to deteriorate. The Hilbert curve gives the best performance for grids with all dimensions. The performance of Gray, Z-scan and the H-scan curves are similar for all grid dimensions.

2. Total Number of Hops :

Analysis : The lesser the number of hops taken by a curve the better its performance is. The Gray and Z-scan curves provide us with the worst results, due to the large number of

jumps taken by them while accessing the grid-cells. The performance provided by the Gray curve, Z-scan curve and H-scan curve does not vary from each other as the size of the grid is increased. The best performance is given by the snake scan curve and the Hilbert curve. As the grid dimensions increase, the difference in performance of the snake curve and Hilbert curve widens as compared to the other three curves.

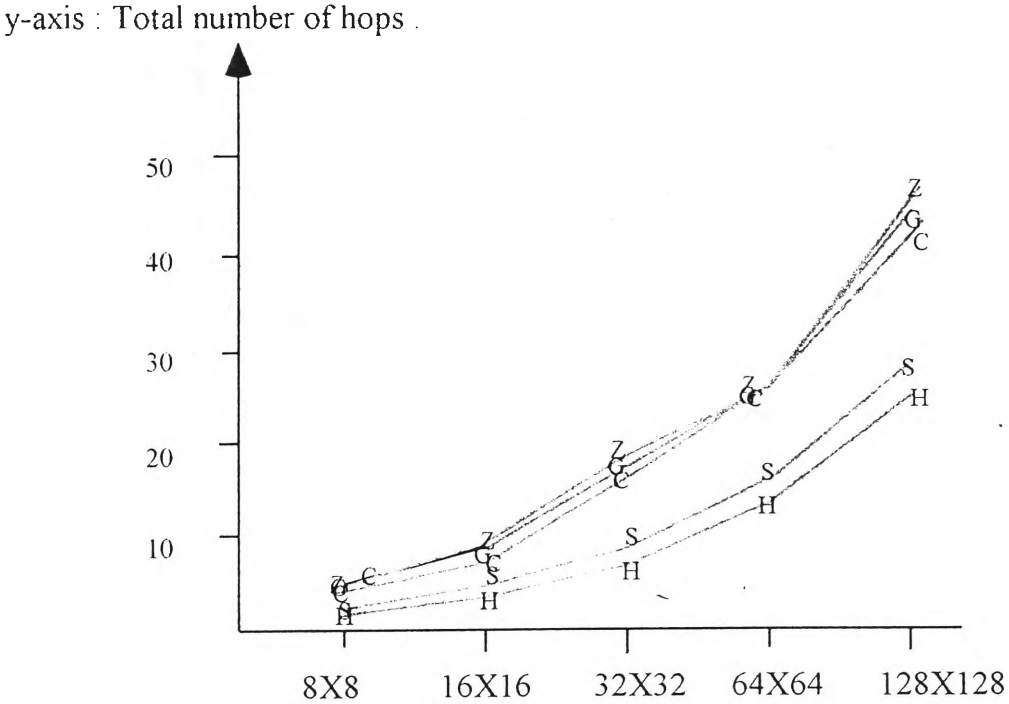


Fig 3.3 Number Of Hops In Exp.2

Curve	8X8	16X16	32X32	64X64	128X128
H	±1	±1	±2	±4	±3
S	±2	±2	±4	±6	±8
C	±1	±2	±2	±6	±10
Z	±1	±1	±3	±3	±6
G	±1	±3	±4	±5	±7

Table 3.8 Error Table for Fig. 3.3

3. Linear Span :

y-axis : Linear span

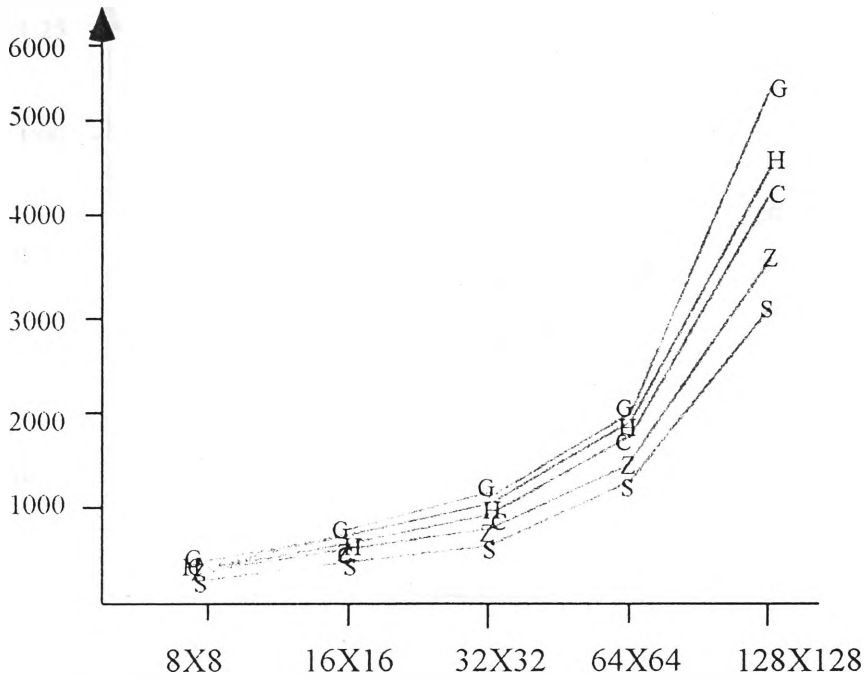


Fig 3.4 Linear Span In Exp.2

Curve	8X8	16X16	32X32	64X64	128X128
H	±3	±5	±7	±6	±8
S	±3	±4	±4	±8	±12
C	±4	±3	±7	±9	±11
Z	±2	±5	±8	±5	±7
G	±3	±2	±6	±6	±11

Table 3.9 Error Table for Fig. 3.4

Analysis : When applied to grids with low dimensions the curves have a similar linear span, and there is a steady rise as the size of the grid increases. The Gray curve gives the worst performance while the snake scan curve gives the best performance for grids with all dimensions. The Hilbert scan, Z-scan and H-scan give an average performance throughout.

4. Fraction of Blocks Read Sequentially :

y-axis : Fraction of blocks read sequentially .

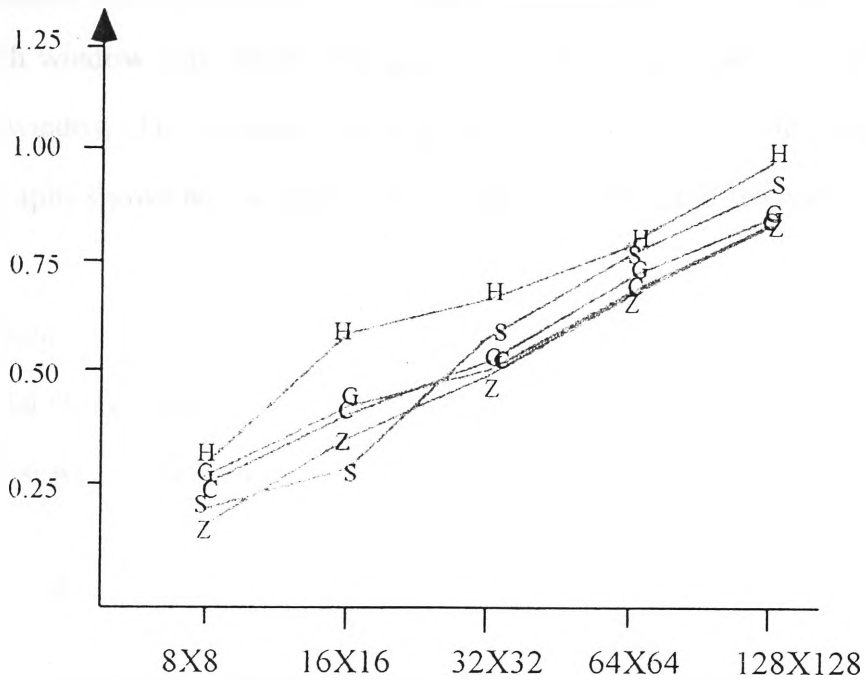


Fig 3.5 Mean Number Of Sequential Blocks Accessed in Exp.2

Curve	8X8	16X16	32X32	64X64	128X128
H	±0.00	±0.01	±0.03	±0.04	±0.03
S	±0.01	±0.01	±0.03	±0.03	±0.05
C	±0.01	±0.02	±0.04	±0.07	±0.09
Z	±0.00	±0.01	±0.02	±0.04	±0.04
G	±0.01	±0.02	±0.04	±0.03	±0.04

Table 3.10 Error Table for Fig. 3.5

Analysis : The Gray curve, H-scan curve and Z-scan curves perform similarly for grids of bigger dimensions. Performance of the snake curve is not so good for smaller dimensions grids, it picks up and gives the second best performance for bigger size grids. The Hilbert curve gives the best performance for grids with all dimensions.

3.1.2.3 Sub-division 3 :

For experiments conducted in this sub-division, the grid dimension was kept constant while the area of search window was varied. The grid dimension was 128X128 units. The area of the search windows lay between the range of 0-16, 16-32 , 32-48, 48-64 and 64-96 units The graphs shown below present mean values of thousand readings.

1. Sequential Block Count :

   y-axis : Sequential block count .

   x-axis : range of search window area .

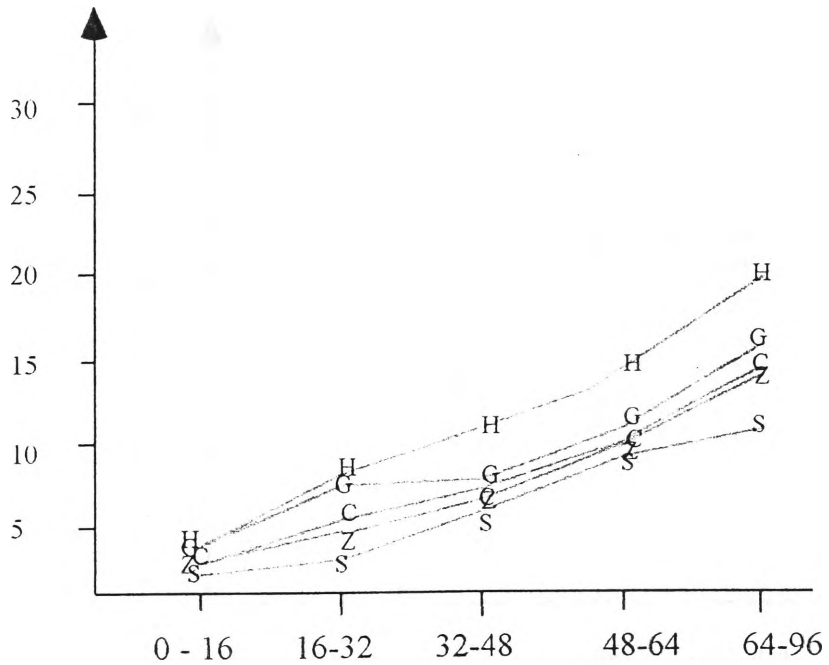


Fig 3.6 Sequential Blocks Accessed in Exp.3

Curve	0 - 16	16 - 32	32 - 48	48 - 64	64 - 96
H	±2	±2	±3	±2	±4
S	±2	±3	±3	±4	±3
C	±1	±1	±3	±2	±2
Z	±2	±2	±5	±3	±4
G	±1	±2	±4	±4	±3

Table 3.11 Error Table for Fig. 3.6

Analysis : The Hilbert curve gives the best performance among the other space filling curves for different search window areas. There is a steep rise in the performance of the Hilbert curve as the search window area increases. The other four curves give a somewhat similar performance, among them the worst performance is given by the snake scan curve. The performance of snake curve begins to deteriorate for search windows with large areas.

2. Total number of hops :

y-axis : total number of hops .

x-axis : range of search window area .

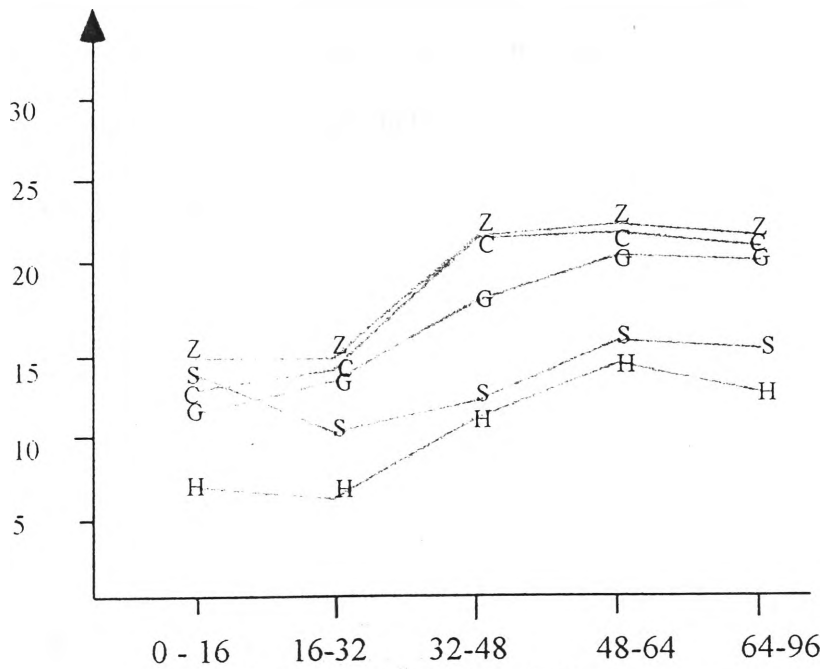


Fig 3.7 Number Of Hops In Exp.3

Curve	0 - 16	16 - 32	32 - 48	48 - 64	64 - 96
H	±2	±3	±2	±4	±3
S	±1	±2	±1	±4	±2
C	±2	±3	±3	±2	±4
Z	±1	±2	±2	±2	±3
G	±1	±2	±3	±3	±4

Table 3.12 Error Table for Fig. 3.7

Analysis : The number of hops taken by the Z-scan curve and H-scan curve are similar for search windows with different areas. Both of these curves perform badly as compared to the other curves. The snake scan curve gives an average performance, better than the Gray curve. The Hilbert curve gives the best performance. The performance of the Hilbert curve is very good for smaller area search windows as compared to the other space filling curves.

3. Linear Span :

Analysis : In this experiment of measuring the linear scan of a curve the worst performance is given by the Gray scan. The number of grid-cells accessed to fetch the grid-cells inside a search window are excessive in the case of a Gray curve as compared to the other curves. The snake scan curve performs the best. As the search window area increases the performance of the snake curve also gets better.

y-axis : linear span.

x-axis : range of search window area .

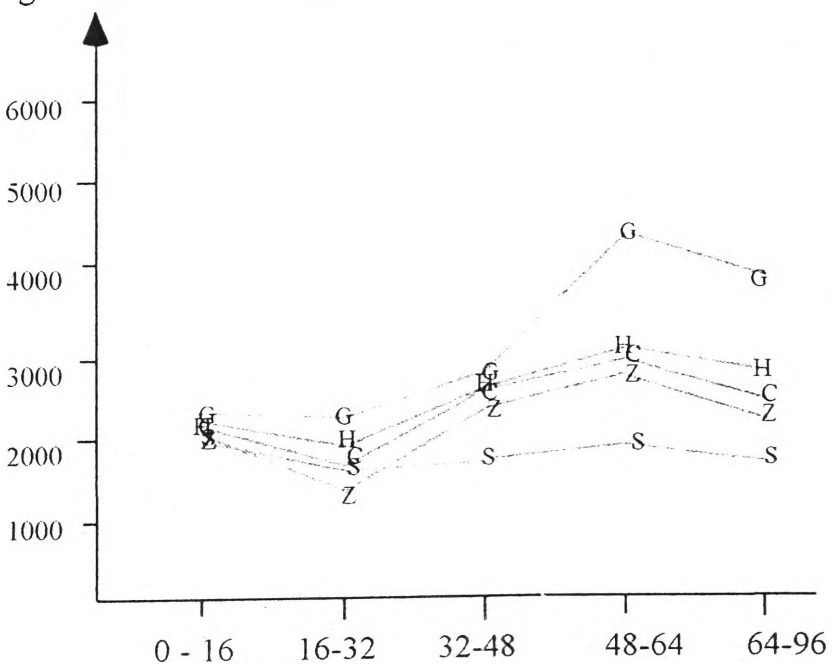


Fig 3.8 Linear Span In Exp.3

Curve	0 - 16	16 - 32	32 - 48	68 - 64	64 - 96
H	±4	±4	±3	±3	±6
S	±3	±5	±5	±7	±6
C	±4	±4	±3	±6	±6
Z	±3	±5	±4	±1	±4
G	±2	±4	±5	±4	±7

Table 3.13 Error Table for Fig. 3.8

4 . Fraction of Blocks Read Sequentially :

y-axis : fraction of blocks read sequentially .

x-axis : range of the search window area .

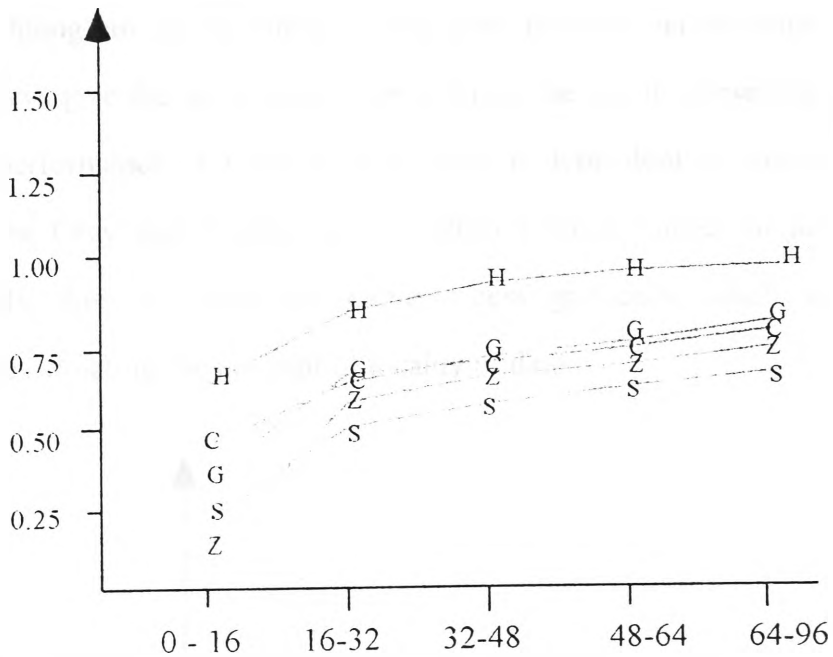


Fig 3.9 Mean Number Of Sequential Blocks Accessed in Exp.3

Curve	0 - 16	16 - 32	32 - 48	48 - 46	64 - 96
H	±0.06	±0.05	±0.08	±0.10	±0.09
S	±0.03	±0.06	±0.06	±0.07	±0.08
C	±0.07	±0.08	±0.04	±0.06	±0.06
Z	±0.04	±0.02	±0.06	±0.03	±0.04
G	±0.07	±0.04	±0.08	±0.08	±0.07

Table 3.14 Error Table for Fig. 3.9

Analysis : The Hilbert curve gives by far the best performance when we measure the mean number of cells accessed sequentially. The performance of the other space filling curves undergoes a steady rise as the search window area increases.

3.1.3 Conclusions

From the various experiments conducted on the space filling curves we can arrange them in order of their performance.

Among the space filling curves the Hilbert curve gives the best performance while the Gray and Z-scan curve give the worst performance. From the results presented above we can infer that the performance of a space filling curve is dependent on maintaining the *locality of data*. The Gray and Z-scan curve undergo a large number of jumps while accessing grid cells, due to which the curve access grid-cells which are not in neighbourhood, thus violating the concept of locality of data.

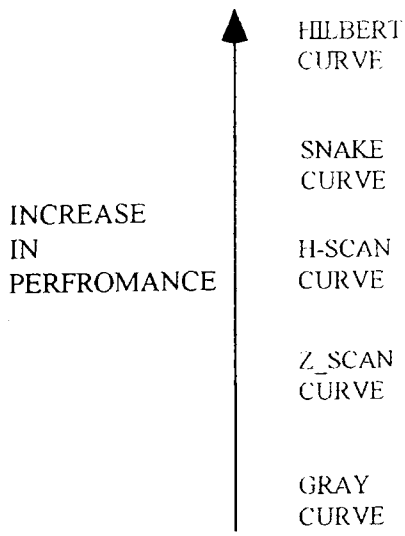


Fig 3.10 Comparison Results

3.2 Summary

In this chapter, we experimentally evaluated the performance of the space filling curves. On basis of performance-results obtained, it was concluded that the Hilbert curve gave the best performance among all space filling curves.

In the next chapter, we consider the case of non uniform distribution of data in two dimensional space. We propose an algorithm for linearly mapping such non uniform distributed data onto a database.

CHAPTER 4

NON REGULAR GRIDS

Up till now, we have assumed uniform density of data objects in two dimensional space, which when using a regular grid allows us to map data objects onto the database efficiently. In this chapter, we consider the case of non-uniform density of data objects in space. We introduce the concept of non-regular grids and present an algorithm useful in mapping non-regular grid-cells onto the database.

4.1 Introduction

So far, we have assumed that data objects are distributed with uniform density throughout the two dimensional space. With this assumption, when a regular grid was applied, each grid-cell contained approximately the same number of data. On mapping regular grid-cells onto the disk blocks we obtained a database with an even distribution of data.

It is not always the case that data objects are uniformaly distributed in space. Applying a regular grid to such a space with non-uniform density data objects (fig 4.1) and mapping the grid-cells onto the database, some disk blocks have an overflow of data while some remain totally empty. To overcome such haphazard distribution of data in a database we propose the concept of non-regular grids .

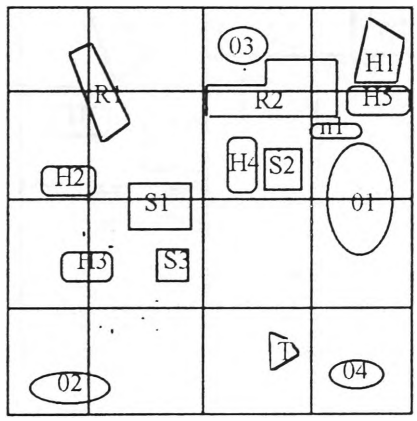


Fig 4.1 Square Tessellations Applied To
Non-Uniform Distribution Of Data

4.2 Non Regular Grids

Non-regular grids provide an efficient way of dividing two dimensional space consisting of non-uniform density of data objects into smaller regions such that each sub-region will contain approximately the same number of data objects. Each non-regular grid-cell is rectangular in shape and its dimensions are dependent on the density of data objects. Regions in the two dimensional space where there is a higher density of data objects will contain grid-cells of smaller dimensions while regions with low data object density will

have large size grid cells. Each non-regular grid-cell should contain approximately the same number of data objects within itself.

In figure 4.2, we have applied a non-regular grid onto a two dimensional space with a non-uniform distribution of data.

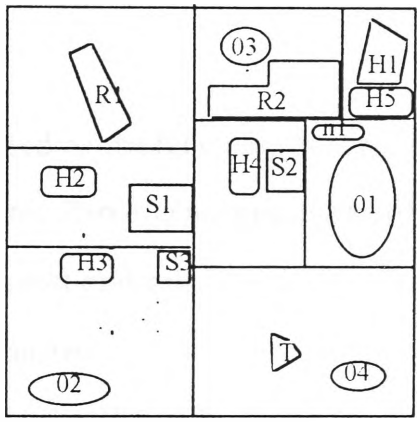


Fig 4.2 Non-regular Tessellations Applied
To Non-Uniform Distribution Of Data.

On mapping the non-regular grid cells linearly onto the database we obtain a uniform distribution of data in the database. Now the main problem arises of mapping non-regular grid-cells linearly onto the disk blocks in one-to-one fashion. The space filling curves are not applicable here as they require a fixed regular grid for linearly mapping data objects. We propose an algorithm for generating a non-regular curve which will help in mapping non-regular grid-cells onto a database. The non-regular curve generated by the algorithm is called *non-reg curve*.

4.3 Non-regular algorithm

In this section, we propose an algorithm for generating a curve called the non-reg curve, which will help in saving non-uniform density two dimensional data in a onto a database.

The algorithm is based on the following assumptions:

(i) The non-reg curve generated from the algorithm starts by selecting grid-cells from the lower left-most grid cell. The curve selects grid-cells until all grid-cells in a non-regular grid are exhausted.

(ii) As discussed earlier, efficient linear clustering is obtained when locality of data is maintained during mapping two dimensional data objects onto a database. Neighbouring data in two dimensional space should be close together in one dimensional space. The algorithm is based on the same idea of maintaining locality of data. The non-reg curve tries to access all the neighbouring grid-cells before accessing other grid cells.

(iii) On basis of the experiments conducted on the space filling curves we reached the conclusion that the Hilbert curve gave the best performance. Our non-reg curve is based on the shape of the Hilbert curve. On applying the non-reg curve to a regular grid, the curve will access grid-cells in the same sequence as the Hilbert curve. We can say that we have extended the Hilbert curve for non-regular grids, along with some additional characteristics.

(iv) The non-reg curve avoids jumps as far as possible. The curve selects grid-cells in a such a sequence so that there are a minimum number of jumps taken.

(v) The algorithm generates the non-reg curves in such a way as to avoid *isolated* grid-cells. A grid-cell is isolated when it has no neighbouring grid-cells in the grid. If selection of the next grid-cell from the current grid-cell results in any grid-cell to become isolated, then the curve will access that grid-cell before accessing the chosen next grid-cell.

(vi) If at any time the curve is left with no neighbouring tile to be selected, then it access a non-neighbouring grid-cell in the grid. The next grid-cell to be selected during a jump is selected such that it is situated nearest to the current grid-cell. If more than one grid-cell is situated at the same distance to the current cell, the next cell is selected randomly from the equal distant grid cells.

(vii) In the algorithm, once a grid cell is accessed by the curve, then it is marked as "traversed" and it is not considered again to be accessed.

The algorithm is based on seven basic steps. These steps are used in combination with each other for generating the non-reg curve. The seven steps are as follows -

(i) Upward Move

-this step traverses only neighbouring grid-cells which are above and on the right hand side to the current cell.

-the first step is to select a neighbouring grid-cell which is above the current cell and left-most as compared to the other neighbouring grid-cells. This selected cell is called as "start-cell".

-the next step is the selection of a grid-cell which is on the right hand side or below the start cell and which is a neighbour to the start cell as well to the current cell. Grid-cells below the current cell are not taken into account. After selecting such a grid-cell, traverse that grid-cell and name it as the start-cell . Mark the previous start-cell as traversed.

-the above step is performed until there are no more grid-cells to be traversed .

-the start cell is now made the current cell and the previous current cell is marked as traversed.

(ii) Right Side Move

-this step traverses only neighbouring grid-cells which are on the right hand side or above the current cell.

-the first step is to select a neighbouring grid-cell which is on the right of the current cell and down-most as compared to the other neighbouring grid-cells. This selected cell is called as "start-cell".

-the next step is the selection of a grid-cell which is above or left of the start cell and which is a neighbour to the start cell as well to the current cell. Grid-cells on the left of the current cell are not taken into account. After selecting such a grid-cell, traverse that grid-cell and name it as the start-cell . Mark the previous start-cell as traversed.

-the above step is performed until there are no more grid-cells to be traversed .

-the start cell is now made the current cell and the previous current cell is marked as traversed.

(iii) Down Move

-this step traverses only neighbouring grid-cells which are below or on the left hand side of the current cell.

-the first step is to select a neighbouring grid-cell which is below the current cell and right-most as compared to the other neighbouring grid-cells. This selected cell is called the "start-cell".

-the next step is the selection of a grid-cell which is to the left of or above the start cell and which is a neighbour to the start cell as well to the current cell. Grid-cells above the current cell are not taken into account. After selecting such a grid-cell, traverse that grid-cell and name it as the start-cell . Mark the previous start-cell as traversed.

-the above step is performed until there are no more grid-cells to be traversed

-the start cell is now made the current cell and the previous current cell is marked as traversed.

(iv) Left Side Move

-this step traverses only neighbouring grid-cells which on the left of or below the current cell.

-the first step is to select a neighbouring grid-cell which on the left of the current cell and upper-most as compared to the other neighbouring grid-cells. This selected cell is called as "start-cell".

-the next step is the selection of a grid-cell which is below or to the right of the start cell and which is a neighbour to the start cell as well to the current cell. Grid-cells on the right to the current cell are not taken into account. After selecting such a grid-cell, traverse that grid-cell and name it as the start-cell . Mark the previous start-cell as traversed.

-the above step is performed until there are no more grid-cells to be traversed .

-the start cell is now made the current cell and the previous current cell is marked as traversed.

(v) Single Cell Move

-this step helps to traverse only a single neighbouring grid-cell in any one of the above mentioned four directions ie upward, downward, right-side and the left-side. Selection of a grid-cell in any direction can be done by performing the second step of the above mentioned four algorithm-steps for a specific direction. Depending on the direction chosen there are four functions present namely cell_upward() , cell_down() , cell_r_side() and cell_l_side(). The new cell traversed is called as the current cell, and is marked as traversed.

(vi) Jump to Nearest Cell

-this steps is performed when there are no neighbouring grid-cells available to be traversed.

-we take into account the four co-ordinates of the current cell . Comparing these co-ordinates with the co-ordinates of all the remaining cells, select the grid-cell which is nearest to the current cell. Make this newly selected grid-cell as the current cell and mark it as traversed . If more than one grid-cell is the same distance from the current cell, then randomly select one of them.

(vii) Check If Alone

-this step is executed after every cell traversal is made . This step checks if there exists any neighbouring grid-cell all of whose other neighbours are traversed, besides the

current cell neighbour ie. if the current cell is traversed then we are left with a grid-cell in memory which has no immediate neighbours .

- When we come across such a grid-cell then that grid-cell is traversed along with the current cell and is marked as traversed .

A combination of the above six steps will generate a curve which helps in saving the non-regular grid cells onto the database records. The complexity of the non-regular algorithm is *O(n)* , where n is the number of grid-cells in a non-regular grid. The algorithm is presented in Appendix-A.

4.3.1 "Non-Reg" Curve

In this section, we provide an example for the non-reg curve. Consider an arbitrary non-regular grid (fig 4.3) and on applying the algorithm to that grid we obtain the non-reg curve.(fig 4.4)

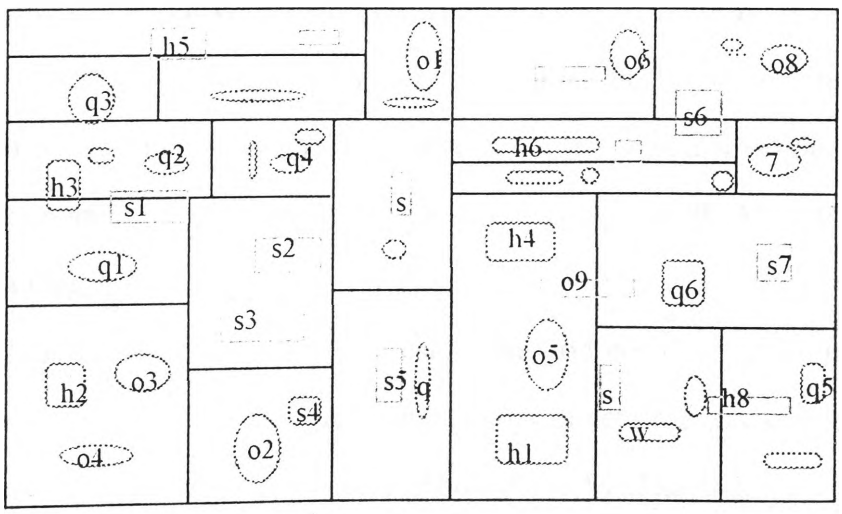


Fig 4.3 Non-regular Grid

The path taken by the non-reg curve is shown using arrows.

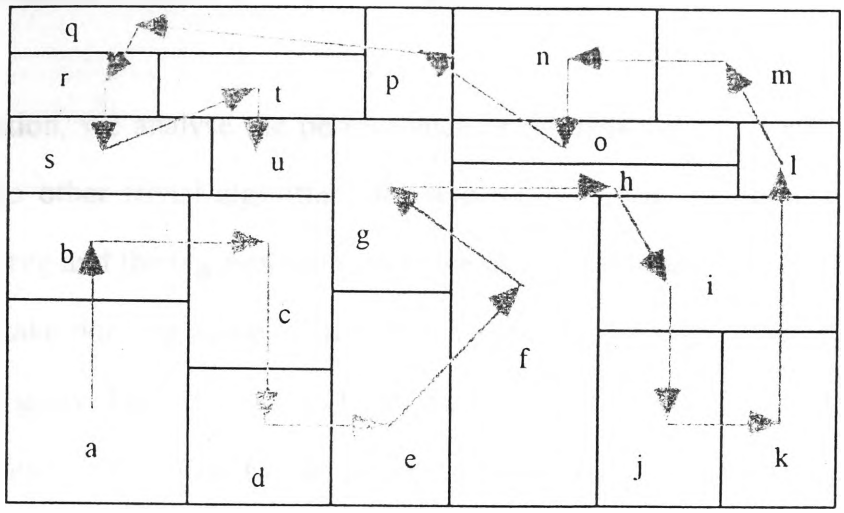


Fig 4.4 Non-reg Curve

As seen from figure 4.4, the non-reg curve tries to access the neighbouring grid-cells before it tries to access other grid-cells. The curve starts from grid-cell 'a', which is the lower left most grid-cell in the two dimensional non-regular grid. After accessing grid-cell 'a' the curve next traverses grid cells 'b', 'c' and 'd' which are neighbours to grid-cell 'a'. Similarly to the Hilbert curve, the non-reg curve tries to traverse all its neighbouring grid-cells before accessing other grid-cells. There is a jump in the curve from grid-cell 'k' to cell 'l'. After traversing grid cell 'k' the curve is not left with any neighbouring grid-cell . The next move must be a jump to the nearest situated grid-cell. After accessing grid cell 'k' the remaining grid cells in the grid cells are 'l', 'm', 'n', 'o', 'q', 'r', 's', 't', and 'u', out of which grid-cell 'l' is situated nearest to the grid cell 'k'. So the next cell selected by the curve is grid cell 'l'. The curve changes its direction every couple of moves in a recursive manner so as to access the neighbouring cells. The steps of the algorithm are provided in Appendix-A.

4.4 Performance analysis of the non-reg curve

In this section, we analyse the performance of the non-reg curve. For comparison, we derived two other trivial algorithms for non-regular grids. The two algorithms, namely snake-non-reg and the big-non-reg curve, are briefly described below:

- Snake-non-reg curve : This curve is defined on similar lines to the snake curve for regular grids. The curve starts from the lower left corner rectangular cell of the grid and traverses cells similar to the snake scan curve. It access grid cells in a vertical sequence. If there is no neighbouring cell existing then the curve selects the closest cell as the next cell. The figure drawn below(fig 4.5) shows the shape of a snake-non-reg curve for a non-regular grid.

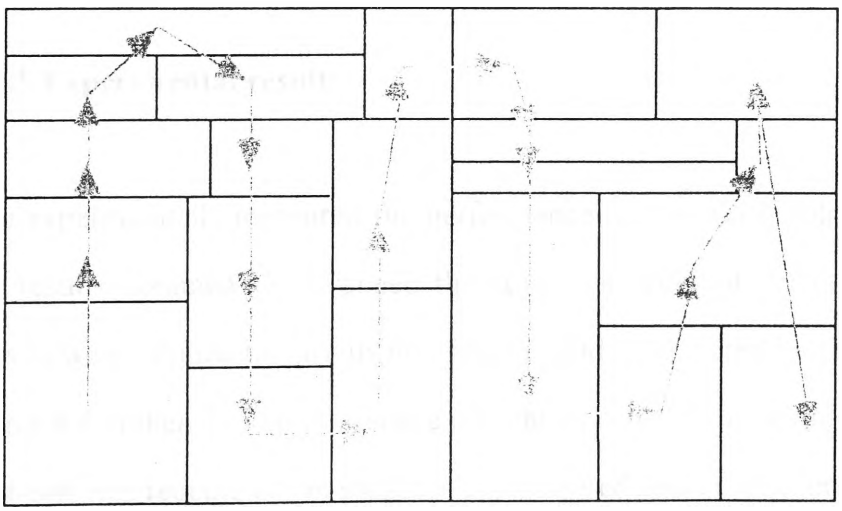


Fig 4.5 Snake-non-reg Curve.

- Big-non-reg curve : This curve too starts from the lower left corner of the grid. The next cell it selects from a current cell is a neighbouring cell enclosing the maximum area as compared to the other neighbouring cells. If there is no neighbouring cell existing

then the curve selects the closest cell with maximum area. The big-non-reg curve is shown in the figure below (fig 4.6).

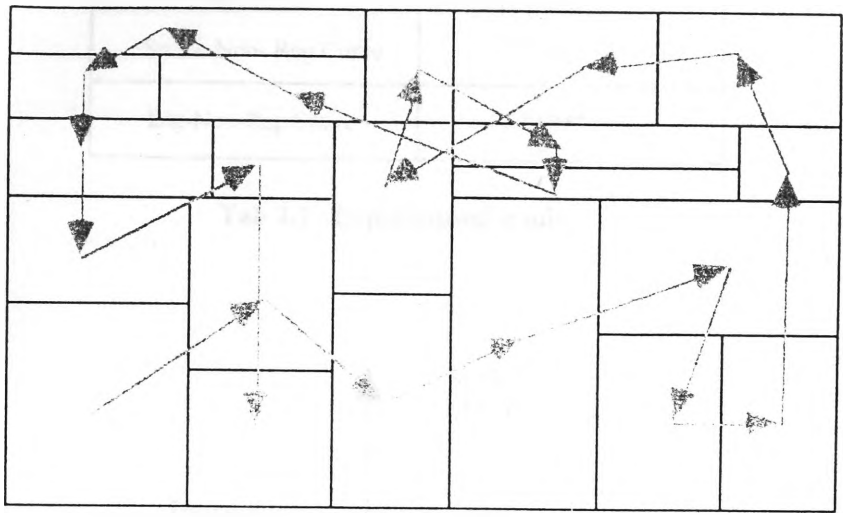


Fig 4.6 Big-non-reg Curve

4.4.1 Experimental results

We experimentally measured the performance of the non-regular curves and on basis of the results obtained we arranged the curves in order of performance. The non-regular curves were applied to various non-regular grids, generated by randomly selecting points is space and then dividing the space into three sub-sections using a point as vertex. Each grid-cell was rectangular in shape. We considered grids consisting of grid-cells within the range 20 to 80 in number. On specifying a search window, the measure of performance was the total number grid-cells accessed before the curve accesses all grid-cells overlapped by the search window. Taking into account the average results of twenty-five readings, we present average result in a table (Tab 4.1) and on basis of the results we have also arranged the curves in increasing order of performance (fig 4.7).

Non - Regular Curves	Avg. No. of Grid Cells Read
Non-Reg Curve	19.132
Snake-Non- Reg Curve	23.442
Big-Non-Reg Curve	30.981

Tab 4.1 Experimental results

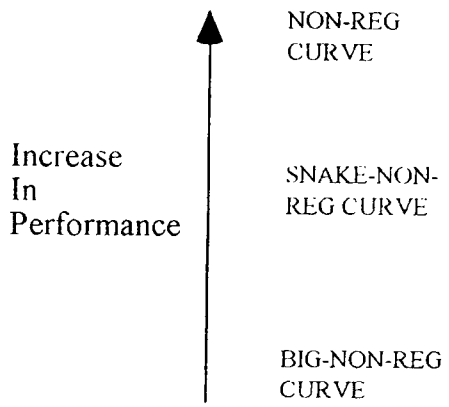


Fig 4.7 Comparison Between Non-regular Curves

The Non-reg curve gives the best performance, while on the other hand the Big-non-reg curve performs very badly due to the haphazard nature in which it access the grid-cells. The Non-reg curve tries to traverse neighbouring grid-cells first before accessing other grid cells, due to which the performance is better when compared to the other two curves. There are a minimum number of jumps taken by the Non-reg curve. In Big-non-reg curve the concept of neighbourhood of cells is not taken into consideration at all. The Snake-non-reg curve gives an average performance, better than the Big-non-reg curve.

4.5 Summary

In this chapter we presented the concept of non-regular grids and proposed an algorithm for linearly saving data from two dimensional space with non uniform data distribution onto a database. We called the curve non-reg curve. The curves performance was measured with two different curves.

In the next chapter we propose an algorithm by means of which we can analytically analyse the performance of a linear clustering curve.

CHAPTER 5

ANALYSIS OF RESULTS

In the previous chapters we analysed the performance of *linear clustering curves* on basis of experiments conducted on them. By linear clustering curves we mean the space filling curves and the non-regular curves. A number of experiments were performed, providing results on basis of which we could arrange the curves in order of their performance. Conducting experiments for evaluating the performance of linear clustering curves is a tedious and time consuming job. In this chapter, we propose an alternative way for analysing a curves performance. An algorithm is presented which helps in analytical analysis of a curve.

5.1 Introduction

So far in this thesis we have analysed the performance of various linear clustering curves on basis of experiments conducted on them. A number of experiments were performed,

under different conditions and using different parameters, on the different curve algorithms. On the basis of the experimental results we could arrange the linear clustering curves in order of their performance.

Conducting various experiments for testing the performance of a curve is a tedious and time consuming job. Each experiments is to be conducted under different conditions and having different parameter values. The performance of a linear clustering curve may not be consistent throughout, under certain conditions the curve may provide optimum performance while not so under different conditions. Instead of performing experiments there should be another way of analysing performance of a linear clustering curve. It should be possible to gauge the performance of a curve by taking a curves *shape* into consideration. In the following section we present an algorithm by means of which we could analytically evaluate the performance of a curve. On applying the algorithm on a linear clustering curve, we can measure its performance.

5.2 Analytical analysis

As discussed in previous chapters, linear clustering of two dimensional data onto a one dimensional space is most efficient when *locality of data* is maintained. Data close together in two dimensional space should also be close together in one dimensional space for efficient clustering of data. We have developed an algorithm using the idea of locality of data as a basis.

Depending on the uniformity of data in the two dimensional space a regular or non-regular grid is applied, segmenting it into rectangular regions called grid-cells. A linear

clustering curve when applied to a grid will save each grid-cell onto a corresponding disk block in a sequence in which the curve access the grid-cell. Efficient linear clustering is obtained when a curve saves a grid-cell along with its *neighbouring grid-cells* in a sequence one after another or as close to each other onto the database A grid-cell is said to be a neighbour to another grid-cell when it has a side or vertex adjacent to the other grid-cells side or vertex. In the figure below (fig 5.1), the neighbouring grid-cells to grid-cell 'A' are 'B','C','D','E','F','G','H',and 'I'. All the other grid-cells in the grid are not neighbours to grid-cell 'A'.

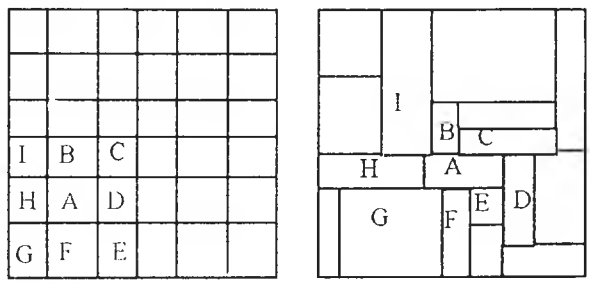


Fig 5.1 Neighbouring Cells;(a) Regular Grids
(b)Non-regular Grids

In the next section, we discuss the assumptions taken into consideration while developing the algorithm.

5.2.1 Algorithm Assumptions

The algorithm developed for analytically analysing performance of linear clustering curves is based on the following assumptions :

(i) The algorithm is applied to each grid-cell in a grid. The algorithm ends execution when all the grid-cells in a grid are exhausted. On selecting a grid-cell, labelled as current grid-cell, the algorithm marks all its neighbouring grid-cells and increments a counter for each grid-cell accessed by a curve until the marked grid-cells are accessed. The measure of performance is the number of grid-cells accessed by a linear clustering curve before all the neighbouring grid-cells to a current grid-cell are accessed.

(ii) In this algorithm the concept of "jumps" plays a major role . A linear clustering curve is said to "jump" when it accesses a non-neighbouring grid-cell from the current grid-cell. The fewer the number of jumps taken by a curve the better its performance is.

(iii) For each grid cell accessed by a linear clustering curve, the algorithm measures the number of grid cells accessed before its neighbouring grids cell are accessed.

(iv) On accessing all the marked grid-cells for a current grid-cell, the number of grid-cells accessed and the total number of jumps taken by a curve are summed up in variables, count and count_jump respectively. Exhausting all the grid-cells in a grid, the performance of the curve is based on the mean value of the variables.

5 .2.2 Algorithm

In this section, we present an algorithm, useful in analytically analysing performance of the linear clustering curves.

Algorithm

(1)**Begin**.
(2)Initialise count-X , count-Y to zero .
(3)Initialise count, count_jumps, sum_count, sum_jumps ,
no_of_selected_cells to zero .
(4)while (the algorithm is not applied for all the grid cells in a grid)
(5)**begin**
(6) Select current grid cell from the grid .
(7) Mark all the neighbouring cells to the current grid cell .
(8) Variable fst_move contains the direction in which the curve moves initially to reach the next cell in sequence ie up , down ,right & left.
(9) Set fst_move to the movement made from the current cell .
Maintain sum of all moves taken for each grid-cell in variable sum_moves ie. sum_moves = sum_moves + count.
Maintain sum of all jumps taken for each grid-cell in variable sum_jumps ie. sum_jumps = sum_jumps + count_jumps.
(10) While (all marked cells not accessed)
(11) **begin**
(12) /*-- increment number of cells selected from grid --*/
(13) no_of_selected_cells++ ;
(14) Set inside_loop = 'n' ;

(15) /*-- if statement for fst_move "up" or "down" --*/
(16) if (((fst_move = 'up') OR (fst_move = 'down')) AND
(inside_loop = 'n'))
(17) **begin**
(18) steps('up' , 'down').
(19) **end**

(20) /*-- if statement for fst_move "right" or "left" --*/
(21) if (((fst_move = 'right') OR (fst_move = 'left')) AND
(inside_loop = 'n'))
(22) **begin**
(23) steps('right' , 'left')
(24) **end**
(25)**end** /*-- end of while --*/
(26)**end** /*-- all grids accessed-- */

/*-- Calculating mean of valued obtained from all the grid cells in the grid -*/

(27) Calculate the mean of variable "sum_count" with respect to all grid-cells in a grid.
(28) Calculate the mean of variable "sum_jumps" with respect to all grid-cells in a grid.

```
        /*-- functions --*/


(29)steps(move_1, move_2)
(30)begin
(31)        check next move ;
(32)        set inside_loop = 'y' ;
(33)        if (next move is a jump and it access a neighbouring
               block)
(34)        begin
(35)              /*-- count number of jumps --*/
(36)              function_jump(next_move) ;
(37)              check next move from the jumped cell ;
(38)              Set next_move to it;
(39)              /*-- count number of moves--*/
(40)              function_count(next_move) ;
(41)              fst_move = next_move ;
(42)        end
(43)        else if (next_move = move_1) || (next_move = move_2)
(44)        begin
(45)              function_jump(next_move);
(46)              fst_move = next_move;
(47)              /*-- end of current "if" --*/
(48)              break ;
(49)        end
(50)        else
(51)        begin
(52)              function_count(next_move);
(53)              function_jump(next_move);
(54)        end
(55)end


(56)     /*-- to count the number of moved made --*/
(57) function_count(next_move)
(58)begin
```

```
(59)      if (jump has not occurred)
(60)        begin
(61)        if (next_move is parallel to x_axis)
(62)              increment count-X by one ;
(63)        else  /*next move parallel to y_axis*/
(64)              increment count-Y by one ;
(65)        increment count by one ;
(66)        end
(67)end

(68)/*-- to count the number of jumps taken  --*/
(69) function_jump(next_move)
(70)begin
(71)      if (jump has occurred)
(72)      begin
(73)         if (jump is parallel to x-axis)
(74)         cnt_jump =   (number of edges parallel to the x-axis lying
                    between the current cell and the next cell jumped to).
(75)      else if (jump is  parallel to y-axis)
(76)         cnt_jump =  (number of edges parallel to the y-axis lying
                      between the current cell and the next cell jumped to).
(77)      else /*jump neither parallel to the x-axis or y-axis*/
(78)      begin
(79)         count   number of grid  edges parallel to the y-axis lying
             between the current cell and the next cell jumped to
(80)         assign the count to cnt_jmp_y ;
(81)         count   number of grid edges parallel to the x-axis lying
             between the current cell and the next cell jumped to.
(82)         assign the count to cnt_jmp_x ;
(83)         cnt_jump =  ( cnt_jmp_x + cnt_jmp_y) ;
(84)    end
(85)         count_jump = count_jump + cnt_jump ;
(86)end
(87)end
(88)END
```

The algorithm consists of mainly two subsections, the section which consists of the 'while' loop (lines 4 - 26) and the section which contain the functions (line 29 - 87). The earlier section contains the 'loop' which is applied till all the grid-cells in the grid are exhausted. The while loop calls the function steps() which in turn calls the functions

unction_count() and function_jump(). The functions function_count() and function_jump() naintain the count of the number of grid-cells accessed and the number of jumps taken espectively. After accessing all the grid-cells in the grid the algorithm calculates the average 'alues (lines 27 - 28), on basis of which we can decide the performance of a curve.

5.3 Analysis of the algorithm

In this section, we present analytical results for the linear clustering curves and compare he results with the experimental results obtained from previous chapters.

Space Filling Curves:

Space Filling Curve	Mean Number of Grid-Cells Accessed	Mean Number of Jumps Taken
Hilbert Curve	15.41	0
Snake Curve	17.98	0
Gray Curve	21.33	14.5
Z-Scan Curve	14.71	66
H-Scan Curve	16.54	6

Table 5.1 Analytical Results for Space Filling Curves

Non-Regular Curves:

Non-Regular Curves	Mean Number of Grid-Cells Accessed	Mean Number of Jumps
Non-Reg Curve	24.98	6.4
Snake-Non Reg Curve	27.81	5.31
Big-Non-Reg Curve	39.11	14.65

Table 5.2 Analytical Results for Non-Regular Curves

Presented below are the comparison results for regular and non-regular grid curves.

5.3.1 Regular Grid Curves

A regular grid is applied to two dimensional space which has a uniform distribution of data. We compare the analytical and experimental results for Hilbert curve, z-scan curve, h-scan curve, Gray curve and the snake curve. In figures drawn on the next page, (fig 5.2 and 5.3) we have arranged the space filling curves in order of their performance.

Analytical Analysis :

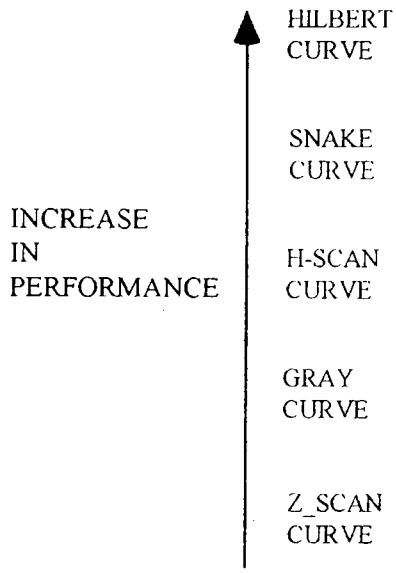


Fig 5.2 Analytical Analysis Of Space Filling Curves

From figures 5.2 and 5.3, we can conclude that the analytical analysis are consistent with the experimental analysis. The Hilbert curve which performed best experimentally also provides us with the best performance when analysed analytically. The other curves give an average performance similar to each other. Analytically the performance of a curve decreases as the number of jumps in a curve increases.

Experimental Analysis:

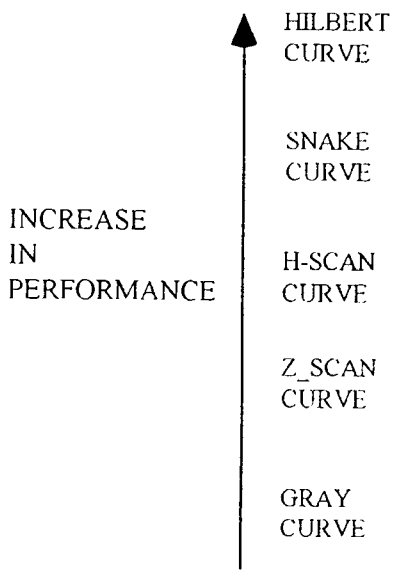


Fig 5.3 Experimental Analysis Of Space Filling Curves

The main inconsistency between the two results is the performance of the z-scan curve. The z-scan curve gives an average performance when computed experimentally but gives the worst performance when the performance is calculated analytically. The reason for this inconsistency is the large number of jumps the z-scan curve takes. The z-scan curve accesses non-neighbouring grid-cell in each alternate step. For the other space filling curves the performance of the curves when calculated experimentally and analytically are consistent to each other.

5 .3.2 Non Regular Grid Curves

As discussed in the previous chapter, non-regular grids are applied to two dimensional space where there is a non-uniform distribution of data. We proposed an algorithm for generating the non-reg curve and also proposed two curves, namely snake-non-reg curve and big-non-reg curve, for comparing the performance with non-reg curve. In the two figures drawn below (fig. 5.4 and 5.5) we have ordered the non-regular curves in order of their performance.

Analytical results

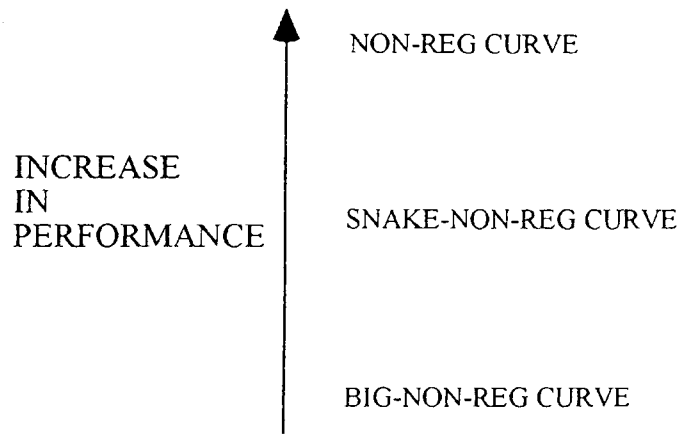


Fig 5.4 Analytical Analysis of Non Regular Curves

Experimental results

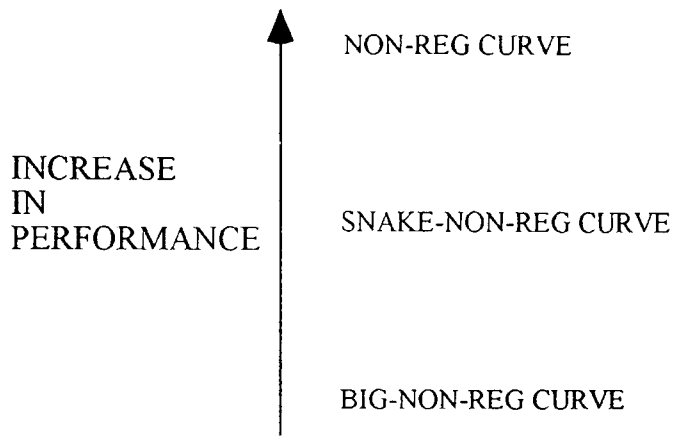


Fig 5.5 Experimental Analysis of Non Regular Curves

From the above two figures, we can infer that the results obtained from the analytical and experimental analysis of a curve are consistent with each other. In non-reg curve, the shape of the curve is such that it tries to access the maximum number of neighbouring cells in a sequence as it can, providing us with the best performance. The big-reg curve, which has no fixed pattern provides us with the worst result analytically as well as experimentally. The curve access the grid-cell with the greatest area next , so the neighbourhood concept is not followed. The snake-non-reg curve gives an average performance with respect to both the analytical and experimental analysis.

5.4 Summary

In this chapter we presented an analytical algorithm for analysing the performance of the linear clustering curves . The algorithm is based on the idea of "locality of data" .The

measure of performance was the number of grid cells accessed before all the neighbouring cells are accessed.

An comparison was made between of the experimental results and the analytical results of the performance of the linear clustering curves . It can be concluded that there is consistency between the experimental and the analytical results obtained .

In the next chapter, we present a brief summary about the thesis and also mention some avenues for further research.

CHAPTER 6

Conclusions

In this thesis, an attempt was made to solve the problem of representing two dimensional data linearly onto a database. An algorithm was presented which helped in linearly saving non-uniform density data from a two dimensional space onto a database. We also analysed some linear clustering techniques. On the basis of results obtained we were able to arrange the linear clustering curves on basis of their performance.

6.1 The linear clustering curves

One of the objectives of this thesis was to extend the technique of linear clustering to two dimensional space consisting of non uniform data distribution. An algorithm was proposed to handle such non uniform distribution of data. Positive results were obtained comparing the performance of the algorithm with other trivial algorithms.

On analysing the linear clustering curves for their performance, on basis of a simplified measure and simulation, we confirmed that the performance of a curve is greatly dependent on it maintaining locality of data, during linearly mapping two-dimensional data onto a one dimensional space.

Another conclusion reached was that the performance of the space filling curve does not depend on the shape of the grid-cells in a grid. The curves gave approximately similar performance for square, hexagonal and triangular grids. The Hilbert curve and snake curve gave the best performance among all other space filling curves. The new space filling curve, the h-scan curve, proposed in this thesis gave results similar to Hilbert curve.

On basis of the performance of the Hilbert curve for regular grids, we tried to extend the Hilbert curve for non-regular grids in hope that it will give optimum performance in non-regular grids too. The non-reg curve for non-regular grids is based on the Hilbert curve. On analysing the performance of the non-reg curve with other non-regular curves, it gave the best performance.

6.2 Further research

Further research should concentrate on developing an algorithm for saving non-uniform density multi-dimensional data onto a database, using the concept of linear clustering. It is also possible to improve upon the algorithm proposed in chapter 4, so as to decrease its complexity and make it more efficient.

Appendix A

Non-Regular Grid Algorithm

NON-REGULAR GRID ALGORITHM

Algorithm Steps :

(1) Accept and save the co-ordinates of all the tiles present in the search area.

(2) Selecting the starting tile . The starting tile will be that tile which is located at the bottom-most and left-most position i.e tile with one set of co-ordinates equal (0,0). Name this tile as the current tile. Mark this tile as traversed and delete it from the memory .

(3) Initial setting :

(4) curr_set = upward ; the latest move made is stored in the variable curr_set . By default we set it to 'upward'.

(5) prev_dir = right ; the latest one-tile movement made is stored in the variable prev_dir .By default we set it to right'.

(6) count = 1 ; count is required to select the next move . Depending on the value of the count the next move is chosen. On making each move the count is incremented by one and when it reaches 5 it is set to 1 .

(7) first_set = upward ; all the moves made are done as a group of four i.e four moves form a set . This variable stores the latestgroup move made . After four moves this variable is utilised to select the next group move . By default this is set to 'upward.

(8) new_set_cnt = 1 ; new_set_cnt is required to select the next group move . Depending on the value of the new_set_cnt the next group move is chosen . On making each group move the new_set_cnt is incremented by one and when it reaches 4 it is set to 1 .

(9) looping_counter = 1 ; This counter is utilised for choosing the one-tile movement direction after each move made. Depending on the looping_counter the next direction of traversal is chosen .

(10) first_step = 'yes' ; this section will be performed only once i.e the first time this algorithm is invoked . After the first execution this variable is set to 'no" .

(11) Perform steps() until all the tiles in the search area are traversed.

```
(12) Steps()
(13) begin
(14)     if (first_step EQ 'yes')
(15)         step_up() .
(16)     Set first_step to 'no' .
(17)     first_one_tile_move()
(18)     if (count EQ 5)
(19)         Set count to 1 .
(20)     if (curr_set EQ 'upward')
(21)     begin
(22)         if (count EQ 1)
(23)         begin
(24)             increment count by 1 ;
(25)             next_set = 'right';
(26)         end
(27)         if (count EQ 2)
(28)         begin
(29)             increment count by 1 ;
(30)             next_set = 'upward'.
(31)         end
(32)         if (count EQ 3)
(33)         begin
(34)             increment count by 1 ;
(35)             next_set = 'upward'.
(36)         end
(37)         if (count EQ 4)
(38)         begin
(39)             increment count by 1 ;
(40)             new_set().
(41)         end
(42)     end
(43)     if (curr_set EQ 'right')
(44)     begin
(45)         if (count EQ 1)
```

```
(46)            begin
(47)                    increment count by 1 ;
(48)                    next_set = 'upward'.
(49)            end
(49)            if (count EQ 2)
(50)            begin
(51)                    increment count by 1 ;
(52)                    next_set = 'right'.
(53)            end
(54)            if (count EQ 3)
(55)            begin
(56)                    increment count by 1 ;
(57)                    next_set = 'down'.
(58)            end
(59)            if (count EQ 4)
(60)            begin
(61)                    increment count by 1 ;
(62)                    new_set().
(63)            end
(64)    end
(65)    if (curr_set EQ 'down')
(66)    begin
(67)            if (count EQ 1)
(68)            begin
(69)                    increment count by 1 ;
(70)                    next_set = 'left'.
(71)            end
(72)            if (count EQ 2)
(73)            begin
(74)                    increment count by 1 ;
(75)                    next_set = 'down'.
(76)            end
(77)            if (count EQ 3)
(78)            begin
(79)                    increment count by 1 ;
(80)                    next_set = 'right'.
(81)            end
(82)            if (count EQ 4)
(83)            begin
```

```
(84)                    increment count by 1 ;
(85)                    new_set().
(86)            end
(87)        end
(88)        if (curr_set EQ 'left')
(89)        begin
(90)            if (count EQ 1)
(91)            begin
(92)                    increment count by 1 ;
(93)                    next_set = 'down'.
(94)            end
(95)            if (count EQ 2)
(96)            begin
(96)                    increment count by 1 ;
(97)                    next_set = 'left'.
(98)            end
(99)            if (count EQ 3)
(100)           begin
(101)             increment count by 1 ;
(102)             next_set = 'upward'.
(103)           end
(104)       if (count EQ 4)
(105)           begin
(106)             increment count by 1 ;
(107)             new_set().
(108)           end
(109) end

(110) if (looping_counter EQ 5)
(111)       Set looping_counter to 1 .

(112) if (next_set EQ 'upward')
(113) begin
(114)       step_up() ;
(115)       if (looping_counter EQ 3)
(116)               tile_1_side .
(117)       else
(118)               tile_upward().
(119)       if (no neighbouring tile found)
```

```
(120)              random_tile().
(121)       increment looping_counter by 1 .
(122) end
(123) if (next_set EQ 'down')
(124) begin
(125)       step_down() ;
(126)       if (looping_counter EQ 3 or 4)
(127)              tile_r_side .
(128)       if (no tile on right side found)
(129)              tile_down().
(130)       else
(131)              tile_down().
(132)       if (no neighbouring tile found)
(133)              random_tile().
(134)             increment looping_counter by 1 .
(135) end
(136) if (next_set EQ 'right')
(137) begin
(138)       step_r_side() ;
(139)       if (looping_counter EQ 3 )
(140)              tile_down .
(141)       else
(142)              tile_r_side().
(143)       if (no tile on right side found)
(144)              tile_down().
(145)       if (no neighbouring tile found)
(146)              random_tile().
(147)       increment looping_counter by 1 .
(148) end
(149) if (next_set EQ 'left')
(150) begin
(151)       step_l_side() ;
(152)       if (looping_counter EQ 3 )
(153)              tile_upward .
(154)             else
(155)              tile_l_side().
(156)       if (no tile onleft side found)
(157)              tile_upward().
(158)       if (no neighbouring tile found)
```

```
(159)              random_tile().
(160)            increment looping_counter by 1 .
(161) end

(162) Set curr_set to next_set .

(163) end /*end of Steps()*/.


/*-- procedure - new_set()--*/
(164) new_set()
(165) begin
(166) if (first_set EQ 'right')
(167) begin
(168)            if (new_set_cnt EQ 1 or 2 )
(169)            begin
(170)              next_set = 'upward' ;
(171)              increment new_set_cnt by 1 .
(172)            end
(173)            if (new_set_cnt EQ 3)
(174)            begin
(175)              next_set = 'left' ;
(176)              increment new_set_cnt by 1 .
(177)            end
(178)            if (new_set_cnt EQ 4)
(179)            begin
(180)              next_set = 'right' ;
(181)              set first-set to 'right';
(182)              increment new_set_cnt by 1 .
(183)            end
(184) end
(185) if (first_set EQ 'upward')
(186) begin
(187)      if (new_set_cnt EQ 1 or 2 )
(188)      begin
(189)              next_set = 'right' ;
(190)              increment new_set_cnt by 1 .
(191)      end
(191)      if (new_set_cnt EQ 3)
```

```
(192)     begin
(193)           next_set = 'down' ;
(194)           increment new_set_cnt by 1 .
(195)     end
(196)     if (new_set_cnt EQ 4)
(197)     begin
(198)           next_set = 'right' ;
(199)           set first-set to 'right';
(200)           increment new_set_cnt by 1 .
(201)     end
(202) end
(203) if (first_set EQ 'down')
(204) begin
(205)     if (new_set_cnt EQ 1 or 2 )
(206)     begin
(207)           next_set = 'left' ;
(208)           increment new_set_cnt by 1 .
(209)     end
(210)     if (new_set_cnt EQ 3)
(211)     begin
(212)           next_set = 'upward' ;
(213)           increment new_set_cnt by 1 .
(214)     end
(215)     if (new_set_cnt EQ 4)
(216)     begin
(217)           next_set = 'upward' ;
(218)           set first-set to 'upward';
(219)           increment new_set_cnt by 1 .
(220)     end
(221) end
(222) if (first_set EQ 'left')
(223) begin
(224)     if (new_set_cnt EQ 1 or 2 )
(225)     begin
(226)           next_set = 'down' ;
(227)           increment new_set_cnt by 1 .
(228)     end
(229)     if (new_set_cnt EQ 3)
(230)     begin
```

```
(231)            next_set = 'right' ;
(232)            increment new_set_cnt by 1 .
(233)      end
(234)      if (new_set_cnt EQ 4)
(235)      begin
(236)            next_set = 'down' ;
(237)            set first-set to 'down';
(238)            increment new_set_cnt by 1 .
(239)      end
(240) end
(241) end /*end of new_set() */
```

```
/*-- procedure for step_up() --*/
(242) step_up()
(243) begin
(244)      At any instance ,if there exists no neighbouring tile
           available to be traversed , then execute the
           random_tile() function to select a new  current tile .
(245)      upward move;
(246)      tile_r_side() ;
(247)      right move ;
(248)      tile_upward() ;
(249)      right move ;
(250)      tile_l_side() ;
(260)      down move.
(261) end /* end of step_up()*/
```

```
/*-- procedure for step_r_side() --*/
(262) step_r_side()
(263) begin
(264)      At any instance ,if there exists no neighbouring tile
           available to be traversed , then execute the
           random_tile() function to select a new current tile .
(265)      right move;
(266)      tile_upward() ;
(267)      upward move ;
```

```
(268)      tile_r_side() ;
(269)      upward  move ;
(270)      tile_down() ;
(280)      left  move.
(290) end /* end of step_r_side()*/


/*-- procedure for step_l_side() --*/
(291) step_l_side()
(292) begin
(293)      At any instance ,if there exists no neighbouring tile
           available to be traversed , then execute the
           random_tile() function to select a new current tile .
(294)      left move;
(295)      tile_down() ;
(296)      down move ;
(297)      tile_l_side() ;
(298)      down move ;
(299)      tile_upward() ;
(300)      right move.
(301) end /*end of step_l_side() */


/*-- procedure for step_down() --*/
(302) step_down()
(303) begin
(304)      At any instance ,if there exists no neighbouring tile
           available to be traversed , then execute the
           random_tile() function to select a new current tile .
(305)      down move;
(306)      tile_l_side() ;
(307)      left move ;
(308)      tile_down() ;
(309)      left move ;
(310)      tile_r_side() ;
(311)      upward move.
(312) end /* end of step_down() */
```

REFERENCES

[AHU81] Ahuja, N., "Approaches to Recursive Image Decomposition,"IEEE Pattern Recognisation and Image Processing, 1981, pp 75-80.

[BEC90] Beckmann, N., et al., "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles," Proc. ACM SIGMOD, 1990.

[BEN75] Bently, J., "Data Structures for Range Searching," ACM Comp. Surveys, Vol 11, no 4, 1979, pp 397-410.

[BER91] Berger, M., Rigoustsos, I., "An Algorithm for Point Clustering and Grid Generation," IEEE Transcations On Systems, Man, and Cybernetics, Vol 21, No 5, Sept 1991, pp 1278-1286.

[BUT71] Butz, A. R., "Alternative Algorithm for Hilbert's Space Filling Curve," IEEE Computers, Apr 1971., pp 424-426.

[GOO92] Goodchild, M.F., Shiren, Y., "A Hierarchical Spatial Data Structure for Global Geographic Information Systems," Geographical Models and Image Processing, Vol 4, No 1, Jan 1992, pp 31-44.

[GAR82] Gargantini, I., "An Effective Way to Represent Quadtrees,", Comm. ACM, Vol 25, No 12, 1982, pp 905-910.

[FAG79] Fagin, R., Nievergelt, J., "Extensible Hashing- A Fast Access Method for Dynamic Files,"ACM Trans. on Databases, Sept 1979, pp 315-344.

[FAL87] "Gray Codes for Partial Match and Range Queries," IEEE Software Engineering, 1987.

[GUN91] Gunther, O., Bilmes, J., "Tree Based Access Methods for Spatial Databases: Implementation and Performing

Exualation," IEEE Transactions on Knowledge and Data Engineering, Vol 3, No 3, Sept 1991, pp 342-353.

[GUT84] Guttman, A., "R-Trees: A Dynamic Index Structure for Spatial Searching," Proc. ACM SIGMOD, 1984, pp 47-57.

[JAG90] Jagadish, H., "Linear Clustering of Objects With Multiple Atributes," Proc. ACM SIGMOD, 1990, pp 332-342.

[KRI93] Krishnapuram, R., "A Possibilistic Approach to Clustering," IEEE Transcations on Fuzzy Syatems, Vol 1, No 2, may 1993, pp 98-110.

[NIE84] Nievergelt, J., Hinterberger, H., "The Grid File : An Adaptable Symmetric Multikey File Structure," ACM Transactions on Database Systems, Vol 9, No 1, March 1984, pp 38-71.

[ORE86] Orenstein, J.A., "Spatial Query Processing in an Object-Oriented Database Syatem," Proc. ACM SIGMOD,1986, pp 326-336.

[ORE89] Orenstein, J.A., "Redundany in Spatial Databases," Proc. ACM SIGMOD, 1986, pp 326-336.

[ROB81] Robinson, J. T., "The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes," Proc. ACM SiGMOD, 1981, pp 10-18.

[SAM84] Samet, H., "The Quadtree and Related Hierarchical Data Structures," ACM Computing Surveys, Vol 16, No 2, June 1984, pp 188-253.

[SAM88] Samet, H., "Hierarchical Representation of Collections of Small Rectangles," ACM Computing Surveys, Vol 29,No 4,Dec 1988, pp 271-309.

[SAM89] Samet, H., The Design and Analysis of Spatial data Structures, Addision-Wesley, 1989.

[SEL87] Sellis, T., Roussopulos, N., Faloutsos, C., "The R+Tree: A Dynamic Index For Multi-Dimensional Objects,"Proceeding of the 13th VLDB Conference, Brighton 1987, pp 507-518.

[STR91] Stroustrup, B., The C++ Programming Language, 2nd ed., Massachusetts: Addision-Wesley, 1991.

[WEB85] Webber, R., Samet, H., "On Encoding Boundaries With QuadTrees," IEEE Trans. on Pattern Analysis and machine Intelligence, Vol 6, No 3, May 1984, pp 365-369.